The Dissertation Committee for Taehwan Choi
certifies that this is the approved version of the following dissertation:

Weak and Strong Authentication in Computer Networks

Committee:

Mohamed G. Gouda, Supervisor

Simon S. Lam

Aloysius K. Mok

Yin Zhang

William D. Young

Sandeep Kulkarni

Weak and Strong Authentication in Computer Networks

by

Taehwan Choi, B.S.;M.S.

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

December 2012

To my wife, Jung Hyun, my parents, and my parents-in-laws

Acknowledgments

I really wondered what PhD is and what a PhD would mean to me before I start my PhD. These kinds of questions become a philosophical question and reminded me of the poet, "Road Not Taken", by Robert Frost. As I am finishing my PhD, these kinds of questions become a realistic question because I realize how many people help me to pursue my PhD. Any words can not express how thankful I am.

First of all, I appreciate my advisor, Dr. Mohamed Gouda, for teaching me how to do research and having me grow as a computer scientist. We used to sit down and think together for hours about my research problems. By doing that, I learned how to think thoroughly and communicate with people. These are great lessons for me to be a good researcher.

I appreciate my committee members, Dr. Simon Lam, Dr. Aloysius Mok, Dr. Yin Zhang, Dr. William Young, and Dr. Sandeep Kulkarni for taking the time to read my dissertation, attend my defense and give invaluable feedbacks out of their busy schedule. It is my honor to have these great committee members.

During my PhD years, I had many ups and downs: My father-in-law, Young Dong Mock, passed away and my professor, Young Hee Chang, at Sogang University passed away during these years. I give my sincere condolences and I appreciate their love and support. My son, Edward Woojin Choi, was born during these years and I am so happy to be a father. This is a great gift to me. I was sometimes sad and sometimes happy during these years. In any case, my wife, Jung Hyun Mok, was

always with me and we shared our sadness and happiness together. She is always supportive and she is the one who understands me most. And I can not describe with any words how thankful I am and how blessed I am because of her.

I am grateful to my family. I am fortunate to have great parents. My father encourages me to pursue my PhD and that really motivates me more than anything else. My mother is supportive and is patient and trust me most. Their loves and supports enable me to achieve my PhD. My parents-in-laws are very supportive to pursue my PhD. Without them, I would not be able to finish my PhD. I am indebted to them in many ways and there is no way I can pay back. I am grateful to my brother, Chul Hwan Choi, and my sister, Arum Choi to support and understand me.

My time at the University of Texas at Austin was a great joy for me and Austin is already my second home town. I had great friends and environments in Austin. I am grateful to everything given to me and I have to think how to give back.

Taehwan Choi

The University of Texas at Austin
December 2012

Weak and Strong Authentication in Computer Networks

Publication No. _____

Taehwan Choi, Ph.D.
The University of Texas at Austin, 2012

Supervisor: Mohamed G. Gouda

In this dissertation, we design and analyze five authentication protocols that answer to the affirmative the following five questions associated with the authentication functions in computer networks.

1. The transport protocol HTTP is intended to be lightweight. In particular, the execution of applications on top of HTTP is intended to be relatively inexpensive and to take full advantage of the middle boxes in the Internet. To achieve this goal, HTTP does not provide any security guarantees, including any authentication of a server by its clients. This situation raises the following question. Is it possible to design a version of HTTP that is still lightweight and

yet provides some security guarantees including the authentication of servers by their clients?

2. The authentication protocol in HTTPS, called TLS, allows a client to authenticate the server with which it is communicating. Unfortunately, this protocol is known to be vulnerable to human mistakes and Phishing attacks and Pharming attacks. Is it possible to design a version of TLS that can successfully defend against human mistakes and Phishing attacks and Pharming attacks?

3. In both HTTP and HTTPS, a server can authenticate a client, with which it is communicating, using a standard password protocol. However, standard password protocols are vulnerable to the mistake of a client that uses the same password with multiple servers and to Phishing and Pharming attacks. Is it possible to design a password protocol that is resilient to client mistakes (of using the same password with multiple servers) and to Phishing and Pharming attacks?

4. Each sensor in a sensor network needs to store $n - 1$ symmetric keys for secure communication if the sensor network has n sensor nodes. The storage is constrained in the sensor network and the earlier approaches succeeded to reduce the number of keys, but failed to achieve secure communications in the face of eavesdropping, impersonation, and collusion. Is it possible to design a secure keying protocol for sensor networks, which is efficient in terms of computation and storage?

5. Most authentication protocols, where one user authenticates a second user, are based on the assumption that the second user has an "identity", i.e. has a name that is (1) fixed for a relatively long time, (2) unique, and (3) approved by a central authority. Unfortunately, the adoption of user identities in a network does create some security holes in that network, most notably

anonymity loss, identity theft, and misplaced trust. This situation raises the following question. Is it possible to design an authentication protocol where the protocol users have no identities?

Contents

Chapter 1

Introduction

Authentication protocols are becoming fixed features in the protocol stacks of computer networks, the Internet, and the World Wide Web [9, 27, 50, 68, 70]. These protocols are usually invoked by the two sides of a communication in order to allow each side to check the "true identity," rather than the "claimed identity," of the other side before the two sides start to exchange their data messages.

In this dissertation, we present the designs of five authentication protocols that can be deployed at different layers in the protocol stacks of computer networks, the Internet, and the World Wide Web. These five authentication protocols are named HTTPI [21], TLP [22], TPP [20], BKP [18], and ITY [19]. Each one of these five protocols is intended to replace an existing protocol and is designed to provide stronger authentication guarantees than those provided by the protocol it replaces:

1. HTTPI is intended to replace HTTP [35]. And whereas HTTP does not provide any authentication guarantees, HTTPI provides server authentication guarantees that are similar to those provided by HTTPS [100].

2. TLP is intended to replace both TLS [27] and the standard password protocol. However, the authentication guarantees provided by TLP are stronger than

those provided by both TLS and the standard password protocol since TLP (unlike TLS and the standard password protocol) can defend against Phishing and Pharming attacks.

3. TPP is intended to replace the standard password protocol. However, the authentication guarantees provided by TPP are stronger than those provided by the standard password protocol since TPP (unlike the standard password protocol) can defend against Phishing and Pharming attacks.

4. BKP is intended to replace existing keying protocols in sensor networks [3, 31, 33, 44, 74]. However, the authentication guarantees provided by BKP are stronger than those provided by existing keying protocols because BKP (unlike existing keying protocols) can defend against collusion attacks.

5. ITY is intended to replace existing anonymous communication protocols [15, 16, 99]. However, the authentication guarantees provided by ITY are stronger than those provided by existing anonymous communication protocols because ITY (unlike existing anonymous communications protocols) does not assume that users have identities.

In the remainder of this Chapter, we summarize the five authentication protocols, HTTPI, TLP, TPP, BKP, and ITY as follows.

1.1 HTTPI Protocol

HTTP and HTTPS are two well-known transport protocols in the World Wide Web. These two protocols are two extremes in terms of costs and security guarantees. HTTP is not expensive to use, but provides no security guarantee. On the other hand, HTTPS is expensive to use, but provides three security guarantees of server authentication, message integrity, and message confidentiality. The recent trends

of technology creates a new class of web applications called "open applications." For example, news, social networking, and blogging require server authentication, message integrity, and optional client authentication. However, these applications do not require message confidentiality. Because HTTP does not support any security guarantee, HTTP is not suitable for "open application." Because HTTPS supports message confidentiality as well as server authentication and message integrity, HTTPS is overkill for "open application." Therefore, we design an HTTP with Integrity or HTTPI to support "open application."

In order for web client C to communicate with web server S using HTTPI, C needs to establish an HTTPI session. The established HTTPI session is uniquely identified by two parameters: 1) session id, 2) session key. The session id is sent in the clear in all the HTTPI requests and responses and the session key remains private to web client C and web server S. HTTPI has the three phases of session: 1) Session Establishment, 2) Session Progression, 3) Session Termination. In the session establishment phase, web client C and web server S use a TLS-like protocol to authenticate web server S to web client C and use a password protocol to authenticate web client C to web server S. They establish their HTTPI session by sharing the session id and agreeing on a session key corresponding to the session id. After the HTTPI session is established, web client C can start to send HTTPI request messages to web server S and web server S can start to send HTTPI response messages to web client C. These HTTPI messages are similar to HTTP messages except for three header fields: 1) Session Id, 2) Header Hash, 3) Content MD5. First, session id is shared during the session establishment and is sent in the clear in the HTTPI header. Second, header hash is the hash value of the header fields with the session key. Third, Content MD5 is the hash of the content body. Thus, if the content body is not changed, the hash of the content body does not need to be computed again. At the end of the progression phase, web server S proceeds

to terminate the session by removing the session entry from the session table and tearing down the connection. We call all the exchanged cookies between web client C and web server S over HTTPI as "verifiable cookies." A verifiable cookie is a cookie with one additional field, called the Verifier. The Verifier is computed by hashing the content of the cookie with both a server key and a session key. Thus, the Verifier cannot be changed without knowing the server key and the session key.

HTTPI provides five security guarantees: 1) Authentication of server S, 2) Integrity of Request Message from C to S, 3) Integrity of Response Message from S to C, 4) Integrity of Cookies from C to S, 5) Integrity of Cookies from S to C. First, authentication of server S comes from the fact that C and S use a TLS-like protocol. Second, integrity of request message from C to S and response message from S to C come from the fact that C and S can check whether the messages are modified or not. The messages cannot be modified without knowing the session key, and thus cannot be modified by other than C and S. Third, integrity of cookies from S to C directly comes from integrity of response message from S to C. Fourth, integrity of cookies from C to S can be done by checking the verifier of the received cookie.

If web client C uses HTTP with web server S, C may suffer from server impersonation, message modification, cookie theft, and cookie injection. If web client C uses HTTPI instead with web server S, none of these attacks can succeed. Server impersonation will not occur because HTTPI has the property of authentication of server S. Message modification will not occur because HTTPI has the property of integrity of request and response messages between C and S. Cookie Theft will not occur because HTTPI has the property of integrity of C to S. Cookie Injection will not occur because HTTPI has the property of integrity of S to C.

One may argue that HTTPS can be used instead of HTTPI. However, there are two reasons that HTTPI is better than HTTPS. First, HTTPI is compatible

with middle boxes such as cache proxies and application firewalls, but HTTPS is not. Second, HTTPI outperforms HTTPS.

We implement HTTPI as a module to Apache server and experiment with the performance with the popular three web pages, an Amazon page, a Facebook page, and a New York Times page. We measure network throughput and CPU execution time. The throughput of HTTPI is within 1.2 % from that of HTTP and is about 37 % better than that of HTTPS. The CPU time of HTTPI is within 1.9 % from that of HTTP and is about 23 % better than that of HTTPS. From these results, we conclude that HTTPI is a reasonable design choice to support security and performance at the same time.

1.2 TLP Protocol

Users type their sensitive data into a web page by relying on the HTTPS URL and the yellow security indicator in the URL bar of a browser. However, users are still vulnerable to many attacks. For example, a user attempts to call a particular HTTPS page, but he ends up going to a wrong HTTPS page in three scenarios caused by human mistakes and Phishing and Pharming attacks as follows. First, a user wants to type `https://www.amazon.com`, but he types `https://www.anazon.com` by mistake. Second, a user receives an email having a link connecting to `https://www.amazon.com` apparently, but the user ends up going to a malicious web page `https://www.anazon.com`. Third, if a web site allows a user to visit a web site by HTTP like `http://www.amazon.com`, the user can be redirected to a malicious web site `https://www.anazon.com` by manipulating the DNS of the user. In the above three scenarios, the user wants to visit `https://www.amazon.com`, but ends up going to `http://www.anazon.com`. If the two web sites have the same look and feel, the user does not notice that he is visiting a malicious web site.

There are two ways to counter the three attack scenarios. One is to educate

people to check the URL carefully, but it is very difficult for users to follow the advice every time they log into every web site. The other one is to have a browser check that the page is indeed the one that a user wants to open before it displays an HTTPS web page. Thus, browser B loads an HTTPS page from web site S when and only when the following three conditions hold: 1) User U has requested the page, 2) Web site S has verified that sometime in the past user U has registered and stored his login data 3) Browser B has verified that sometime in the past user U has registered and stored his login data in web site S. If any of the three conditions does not hold, the browser does not load the page.

Based on the three conditions mentioned above, we propose to modify browsers and web sites in three parts. First, browsers need to classify each displayed HTTP web page as white, and each displayed HTTPS web page as either yellow or brown. A user should regard a white page as insecure, a yellow page as mildly secure, and a brown page as highly secure. Second, our new login protocol, the Transport Login Protocol (TLP), needs to be added to browsers and web sites. Third, browsers assign a classification, white, yellow, or brown, to each displayed web page with the following three rules: 1) Any HTTP page is classified by white, 2) Any HTTPS page by TLP is classified by brown, 3) Any HTTPS page by TLS is classified by yellow if this page is called from a white or yellow page, or brown if this page is called from a brown page.

TLP has five features that do not exist in the current login protocols. First, TLP is immune to attacks because user U succeeds in login iff both browser B and web site S verify that user U has registered sometime in the past with web site S. Second, TLP does not use external servers because all the login data are stored in web site S. Third, TLP supports one-time login data because login data are updated after each successful login of user U into web site S. Fourth, user U needs to remember one password, a TLP universal password, because TLP universal pass-

word is used to log into every web site. Fifth, TLP uses only standard cryptography like AES and SHA-1.

TLP achieves the five objectives: 1) Browser B checks that web site S is one of the web sites where user U has registered, 2) Web site S checks that user U enters his universal password, 3) Both browser B and web site S agree on a session key, 4) Browser B selects a new nonce n' and stores the login data in web site S, 5) Web site S sends to browser B the next HTTPS page that browser B needs to visit.

In order to achieve the five objectives, TLP implements the four messages. Initially, user U needs to store a login data with shared secrets. First, browser B sends a hello message to web site S with a user's identity U. Web site S fetches the login data of user U with the user's identity. Second, web site S replies with a hello-reply message with a nonce and the server nonce encrypted with the current shared secret. Browser B learns the server nonce by decryption. Third, browser B sends a login message with a user's identity U, the password for the current login, a browser nonce, and new login data encrypted with a shared secret and a session key. When web site S receives the login message, web site S checks that user U enters its TLP universal password, extracts the browser nonce, and stores a new login data. Finally, web site S replies with a login-reply message with a next HTTPS URL and a cookie encrypted with the session key. Browser B knows that web site S is one of the web sites where user U has registered before.

TLP defends against user impersonation, human mistakes, Phishing and Pharming attacks, and eavesdropping because an attacker will not be able to compute shared secrets needed to log into web site S.

1.3 TPP Protocol

Current online authentication involves a combination of two protocols: Transport Layer Security (TLS) and a simple password protocol, which we have named the

One-Way Password Protocol (OPP). TLS achieves two outcomes: 1) Client C knows that it is indeed communicating with web site S, 2) Both C and S agree on a master secret that they can use to encrypt and decrypt all the messages that they need to exchange next. As a result, the authentication by TLS is not symmetric and web site S does not know the client with whom it is communicating. Thus, TLS is paired with OPP to authenticate client C to web site S. However, this combination does not defend against many attacks like Phishing attacks.

The combination of TLS and OPP has one subtle weakness: authenticating web site S to a user's browser is not the same as authenticating web site S to user U. In order to leverage the weakness, an adversary only needs to make users believe that malicious web server M is associated with web server S by having M show the same graphics and images of S. Essentially, this is how Phishing attacks work. Although web site M is not related to web site S, it is hard for user U to distinguish web site M from web site S. Therefore, we need a way to show that these two web sites belong to two different domains.

The standard practice of OPP and TLS is broken by Phishing attacks because web site S is not properly authenticated to user U. In other words, user U knows that he is communicating with web site M, but has no way of knowing whether or not web site M is associated with web site S with which user U has a relationship before. So we could make the password protocol symmetric: web site S is authenticated to user U by knowledge of a secret called the *server password*. In fact, this Server Password Protocol (SPP) is usually called the Site-Key Protocol and a site key can be a unique image or a phrase. The usual mechanism is for user U to store the server password on web site S and web site S is authenticated to user U by sending the server password. However, this attractive solution cannot solve Phishing attacks as it is still vulnerable to Man-In-The-Middle attacks.

Considering the fact that an adversary can steal passwords from user U using

Phishing attacks, we can think of a way to ensure that the password of user U on web site S cannot be stolen. The Universal Password Protocol (UPP) prevents the password from being stolen by an attacker. User U only has to remember one universal password and the password is computed by hashing the universal password and the domain name of web site S. User U stores the hash of the password in registration with web site S. Even if web site S is compromised, an attacker cannot use the hash of the password to impersonate user U. When a Phishing attack occurs by a malicious server M, user U uses the domain name of M to compute the password for web site M. As a result, M cannot learn the password of user U on web site S. Although this protocol can protect the password of user U on S, UPP does not protect the other credentials like credit card numbers. Thus, this protocol still has the same weakness as SPP.

We address the vulnerability of UPP to develop the Two-Way Password Protocol (TPP). Unlike UPP, TPP ensures that the server password of web site S binds to the domain name. TPP uses the same password like UPP and the password is the hash value of the universal password and the domain name of web site S. However, TPP uses the hash of the password as a server password. Now when a Man-In-The-Middle attack occurs, a malicious web site M needs to compute the hash of the password, but it cannot be computed without knowing the universal password. Thus, TPP ensures that the server password cannot be just passed by the Man-In-The-Middle and the other credentials cannot be stolen, either.

Although TPP achieves the goal of protecting users from Phishing attacks, TPP does not have the feature of one-time login data. We could think of the case where the password is leaked in practice. In such a case, we want to minimize the damage by incorporating one-time login data into TPP: the secrets are updated on each login. Even if an adversary steals the password of U, the stolen password is only useful until the next time user U logs into web site S. We call this protocol

the Dynamic Two-way Password Protocol (DTPP).

TPP defends against Phishing attacks by introducing the hash of password as a server password and TPP binds the communication from web server S to browser B. The password for TPP is the same as the password for UPP and the password is the hash of a domain name and a universal password. Thus, the universal password cannot be stolen by an attacker.

1.4 BKP Protocol

Many wireless sensor networks are deployed randomly and any sensor can end up being with any other sensor. In this environment, any sensor z who attempts to disrupt the communication between sensor x and sensor y is an adversary and sensor z can launch two attacks: 1) impersonation attack, 2) eavesdropping attack. To defend against these attacks, sensor x and sensor y are required to share the secret known only to sensor x and y, but not other sensors.

A keying protocol for a sensor network is to assign a unique symmetric key to each pair of distinct sensors x and y in the network. Each symmetric key is assigned by the keying protocol before the sensor network is deployed. In a trivial approach, every sensor is required to store $n-1$ symmetric keys if there are n sensors. However, this trivial approach is not efficient because it requires the huge number of keys ($O(n^2)$) needs to be stored as n grows.

There are two ways for a senor to know a symmetric key: 1) storage, 2) computation. The cost of a keying protocol for a sensor network is measured by the number of keys that need to be stored in the sensor network. Thus, it is important to reduce the number of keys stored in the sensor network.

Suppose that the number of sensors in our sensor network is n where n is an odd positive integer, without loss of generality. Each sensor is assigned a unique identifier in the range $0 \ldots n-1$. In this network, each pair of two sensors x and y

shares a key to perform two functions: 1) mutual authentication, 2) confidential data exchange. If the shared symmetric keys are designed to have a "special structure," then each sensor needs to store only $(n + 1)/2$ shared symmetric keys. For the "special structure," we define two new concepts: 1) universal keys, 2) asymmetric relation, named *below*. Each sensor x stores a symmetric key, called the *universal key* of sensor x. Sensors x and y have identifiers ix and iy, respectively and identifier ix is said to be *below* identifier iy iff exactly one of the following two conditions holds: 1) $ix < iy$ and $(iy - ix) < n/2$, 2) $ix > iy$ and $(ix - iy) > n/2$. The special structure of the symmetric key is the hash of identifier ix and the universal key uy where ix is below iy. In this case, the symmetric key needs to be stored in sensor x and sensor y can compute it whenever sensor y wants to use it. Thus, each sensor x stores one universal key and $(n - 1)/2$ symmetric keys for every sensor y where ix is below iy and the number of keys stored in sensor x becomes $(n + 1)/2$.

Now if two sensors x and y happen to be adjacent with each other, then they need to execute a mutual authentication protocol to communicate securely with each other. Sensor x selects a random nonce nx and sends a hello message with nx and its identifier ix and sensor y selects a random nonce ny and sends a hello message with ny and its identifier iy. Sensor x and sensor y determine whether their identifiers are below or not. If its identifier is below, it gets the shared key from the storage. Otherwise, it computes the shared key. Finally, sensor x sends a verify message by hashing its identifier ix, the counterpart's identifier iy, and the received nonce ny with the shared key. Similarly, sensor y sends a verify message to sensor x. This mutual authentication protocol defends against impersonation and Man-In-The-Middle attacks.

After two sensors authenticate to each other by the mutual authentication protocol, they can start exchanging data messages using the data exchange protocol as follows. Sensor x sends its identifier ix, the counterpart's identifier iy, and the

encryption of nonce ny and the text of the data message. Similarly, sensor y sends the data to sensor x. This data exchange protocol defends against eavesdropping and replay attacks.

With our keying protocol, every sensor in the sensor network is required to store only $(n+1)/2$ keys and the total number of keys to be stored in the sensor network is $n(n+1)/2$. This is much better than the trivial approach to store $n(n-1)$ keys. One can ask whether there exists a better algorithm to store much less keys than $n(n+1)/2$ keys. Theoretically, each keying protocol, that is collusion-proof, requires the sensor network to store at least $n(n-1)/2$ keys. Thus, our keying protocol is optimal in terms of the number of keys and thus called the Best Keying Protocol (BKP).

There are two problems with our keying protocol: 1) The number of symmetric keys, $n(n+1)/2$, is still large if n is large, 2) If a sensor network is deployed and if later the number of sensors is increased beyond the upper bound n, the stored keys must be changed. To solve these problems, we introduce the concept of a "sensor role" as follows. Each sensor in a sensor network has a role and the role of a sensor describes the task of the sensor. Many sensors in a sensor network can have the same role to provide fault-tolerance. Thus, the number of roles, m, is relatively small and it is easy to add any sensor to existing roles and to add a new role in a sensor network.

Our BKP successfully reduces the number of keys needs to be stored in the sensor network by using the special structure of having one universal key and an asymmetric relation called *below*. Our BKP is not only efficient but also secure against collusion attacks compared to most efficient keying protocols.

1.5 ITY Protocol

Most networks are designed assuming that every user is assigned an *identity*, which satisfies three conditions: 1) Fixed 2) Unique 3) Approved by a Central Authority. This identity plays an important role in a network in three ways: a) User Identification b) User Authentication c) User Reputation. Although this adoption facilitates the execution of a network, it also incurs some security problems like anonymity loss, identity theft, and misplaced trust. There are two approaches to remedy these security holes. One is to defend against each one of these holes; another is to take a completely different approach to design a network without identities. We take the latter just because no one attempted this approach before.

We design a network without identities by replacing identities with an address and a nonempty set of pseudonyms. The pair, an address and a nonempty set of pseudonyms, satisfies three conditions: 1) Not Necessarily Fixed 2) Unique 3) Approved by a Central Authority. Interestingly, the first condition is the opposite of that of an identity. This means that an address and a nonempty set of pseudonyms cannot be used for an identity. Based on this observation, we design the three protocols: 1) registration protocol 2) connection protocol 3) authentication protocol.

The registration protocol allows each user x in the network to register its current address and its current pseudonym set periodically. Every user is required to send a registration message to the network every T seconds. The registration message consists of an address, a nonempty set of pseudonyms, a registration key, a timestamp, and the message signature signed by the private key corresponding to the registration key. When the network receives the registration message, it first checks the timestamp and then the signature. If the timestamp is not "close" to the real time or the signature is not correct, the network discards the registration message. If the network does not find the address that matches the received address in the registration table, then the network adds the tuple of the address, the nonempty set

of pseudonyms, the registration key, and the timestamp. Otherwise, the network updates the address in the existing tuple. The network checks the registration table and discards every tuple that has not been updated for more than $2T$ seconds. There are three cases for the registration table not to be updated: 1) User x has failed or has quit the network 2) User x has changed its address 3) User x has changed its registration key. In all the three cases, the tuples of the registration table need to be removed.

The connection protocol allows two users in the network to become *connected* to one another. This implies that each user knows the current address of the other user and the two users share a symmetric key. The connection protocol consists of three messages: a request message from any user to the network requesting to be connected to another user and two reply messages from the network to the two users notifying that they are connected. When user x wants to establish a connection with another user y, user x sends a request message to the network. When the network receives the request message from user x, it checks the timestamp and the address. If the timestamp is not "close" to the real time or the network does not find any tuple having the same address, then the network discards the message and terminates the protocol. If the network finds two distinct tuples for user x and user y and their pseudonyms are contained in the nonempty pseudonym sets of user x and user y, the network generates a symmetric connection key and sends a reply message to user x and user y by encrypting the symmetric connection key with the registration key of user x and user y, respectively.

After initiating the connection protocol, user x and user y obtain the current address of the other user and a copy of the symmetric connection key. After user x and user y establish the connection and before they start exchanging data messages over the established connection, user x and user y need to execute the authentication protocol. If the connection is not the first one between user x and user y, they will

have agreed on four items in their last established connection: 1) a pseudonym for user x 2) a pseudonym for user y 3) a token for user x 4) a token for user y. These items are stored as a tuple in the authentication table of user x and user y. In order for user x to be authenticated by user y, user x sends the stored token from the authentication table by encrypting with the connection key to the current address of user y. When user y receives the token encrypted with the connection key from user x, user y decrypts the received token and compares with the stored token in the authentication table of user y. If they are matched, user y authenticates user x and user y sends user x the stored token from the authentication table of user y by encrypting with the connection key. Otherwise, user y sends user x a random value as a token. When user x receives the encrypted token from user y, user x compares the received token with the stored token. If they are matched, user x authenticates user y. After user x and user y receive the tokens from each other, they send a new pseudonym and a new token by encrypting with the connection key. If user y or user x is authenticated, user x or user y removes the existing tuple, respectively. In any case, user x and user y insert a new tuple with the new pseudonym and the new token into their authentication table.

These three protocols (registration, connection, and authentication protocol) are designed to defend against many attacks including pseudonym theft, address theft, wrong address, message forging, replay, impersonation, and man-in-the-middle attack. Therefore, we design a network without identities and defend against anonymity loss, identity theft, and misplaced trust.

1.6 Organization

This dissertation is organized as follows. In Chapter 2, we present our design of the HTTPI protocol, which defends against attacks that cannot be defended against by HTTP. In Chapter 3, we present our design of the Transport Login Protocol

(TLP), which defends against Phishing and Pharming attacks. In Chapter 4, we present our design of the Two-Way Password Protocol (TPP), that also defends against Phishing and Pharming attacks. In Chapter 5, we present our design of the Best Keying Protocol (BKP), that can be used over computationally constrained networks like sensor networks. In Chapter 6, we present our design of the ITY protocol, that provides authentication in networks where users have no identities. Finally, Chapter 7 concludes our dissertation and discusses future research.

Chapter 2

HTTPI: An HTTP with Integrity

The World Wide Web famously supports two transport protocols: HTTP and HTTPS. These two protocols are at the opposite ends of three dimensions: security guarantees, cost of use, and compatibility with middle boxes (e.g. cache proxies) in the Internet. At one end, HTTP provides no security guarantees, but it is inexpensive to use, and is compatible with middle boxes in the Internet. At the other end, HTTPS provides three security guarantees, but it is expensive to use and is not compatible with middle boxes. Although the three security guarantees provided by HTTPS, namely server authentication, message integrity, and message confidentiality, are important in general, many web servers (e.g. email servers) do not need the message confidentiality guarantee. In this Chapter, we present a new transport protocol for the Web, named HTTPI. This protocol provides both server authentication and message integrity, but not message confidentiality. Like HTTP, HTTPI is inexpensive to use and is compatible with middle boxes, and like HTTPS, it defends against many cyber attacks (e.g. Pharming attacks) that HTTP cannot defend against. We developed a preliminary implementation of HTTPI and showed

Table 2.1: Security guarantees of HTTP, HTTPS, HTTPA, HTTPI

	Server Auth	Client Auth	Integrity	Confidentiality
HTTP				
HTTPS	✓		✓	✓
HTTPS with Password	✓	✓	✓	✓
HTTPA		✓	✓	
HTTPI	✓		✓	
HTTPI with Password	✓	✓	✓	

through experimentation that the throughput of HTTPI is within 1.2% from that of HTTP and is 37% better than that of HTTPS. HTTPI was published in [21].

In most cases, HTTPS is also augmented with a password protocol in order to provide the added guarantee of client authentication. (Note that HTTP cannot be easily augmented with a password protocol to provide client authentication.)

A third transport protocol for the web, namely HTTP Authentication or HTTPA for short, has also been proposed [37]. The cost of using HTTPA is somewhere in the middle between the two costs of using HTTP and of using HTTPS. HTTPA provides only two security guarantees: client authentication and message integrity [37].

Therefore, depending on the security requirements of some web application, the application designers can choose to deploy their application on top of HTTP, HTTPS, or HTTPA. For example, the designers of some search engine, which requires no security, can deploy their engine on top of HTTP. On the other hand, the designers of some on-line banking application, which requires maximal security, can deploy their application on top of HTTPS after it is augmented with a password protocol.

Recently, however, a new class of web applications, which we refer to as "open applications," has emerged. And it turns out that the security requirements

of open applications do not quite match the security guarantees provided by HTTP, HTTPS, or HTTPA. Examples of these open applications are web email, social networking, and web blogging. The security requirements of these open applications are (1) server authentication, (2) message integrity, and in some cases (3) client authentication. These open applications, however, do not usually require message confidentiality.

Because neither HTTP nor HTTPA provides all the security requirements of open applications, these applications cannot be deployed on top of HTTP or HTTPA. Also, because open applications do not usually require the expensive requirement of message confidentiality, which is provided by HTTPS, deploying these applications on top of HTTPS is both an expensive and overkill proposition. Moreover, as explained below, because HTTPS is not compatible with middle boxes (such as cache proxies) in the Internet, deploying open applications on top of HTTPS prevents these applications from taking advantage of the middle boxes in the Internet.

Thus, in order to support the secure and efficient deployment of open applications in the web, we present in this Chapter a new transport protocol for the web, which we refer to as HTTP Integrity or HTTPI for short. HTTPI provides the two security guarantees of server authentication and message integrity. But it does not provide message confidentiality. Moreover, HTTPI can be augmented by a password protocol in order to provide the added guarantee of client authentication.

For convenience, Table 2.1 lists the security guarantees that are provided by each member of the HTTP family of protocols. Because the security requirements of open applications are server authentication, message integrity, and in some cases client authentication, it is clear from Table 2.1 that these applications are better deployed on top of HTTPI (with or without password protocol).

The rest of this Chapter is organized as follows. In Section 2.1, we introduce the two main features of HTTPI, namely HTTPI sessions and verifiable cookies. In

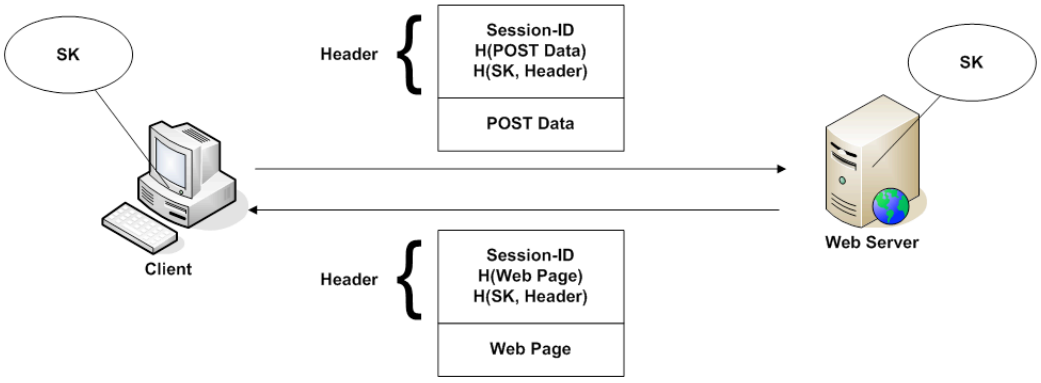


Figure 2.1: Progression Phase in HTTPI

Section 2.2, we formally present the security guarantees provided by HTTPI and discuss how HTTPI provides these guarantees. Then in Section 2.3, we discuss some well-known cyber attacks that HTTPI can defend against but HTTP cannot. And in Section 2.5, we argue that HTTPI is compatible with middle boxes (such as cache proxies and application firewalls) in the Internet but HTTPS is not. In Section 2.6 we present some experimental results that compare the performance of HTTPI against those of HTTP and HTTPS. These results suggest that the throughput of HTTPI is within 1.2% of that of HTTP and is 37% better than that of HTTPS. Finally, we end the Chapter by discussing related work in Section 2.7 and offering some concluding remarks in Section 2.8.

2.1 Design of HTTPI

In order for a web client C (i.e. a browser) to communicate with a web server S using HTTPI, C needs first to establish an HTTPI session with S. The established session is uniquely identified by two parameters:

1. A session id that is computed by S and communicated to C during the session

establishment phase

2. A session (symmetric) key that is computed by both C and S during the session establishment phase

The session id will be sent in the clear in all the HTTPI request and response messages that are exchanged between C and S in the established HTTPI session. The session key will remain private, known only to C and S.

The HTTPI session is established over a TCP connection between a high-numbered TCP port in C and the TCP port 80 in S.

In this Section, we describe the three phases of an HTTPI session between C and S: establishment, progression, and termination. Then, we describe the verifiable cookies which can be exchanged in each such session.

2.1.1 Session Establishment

The web client C and the web server S use a TLS-like protocol to establish their HTTPI session. This protocol allows C and S to perform three tasks.

i. The web client C becomes certain that it is indeed communicating with the right web server S. (Moreover, if a password protocol is added to the TLS-like protocol, then S also can become certain that it is indeed communicating with the right web client C.)

ii. The web server S chooses a unique id for the session and communicates it to C. Server S also chooses a future expiration time for the session.

iii. Both C and S agree on a symmetric session key known only to C and S.

At the end of the session establishment phase, the web client C associates the following session entry with its TCP port for this session:

(Session Id, Session Key, IP address of S, Session Cookies)

The last field "Session Cookies" in this entry is a pointer to all the cookies that are accumulated in C during this session. (See Section 2.1.4 below.) Also at the end of the session establishment phase, the web server S adds the following session entry to its session table:

(Session Id, Session Key, IP address of C, TCP port number in C, Session Expiration Time)

2.1.2 Session Progression

Once an HTTPI session between C and S is established, C can start to send HTTPI request messages to S and receive HTTPI response messages from S. Each HTTPI message, whether a request or response, is similar to an HTTP message with three exceptions:

a. The header of each HTTPI message has a new field, called Session Id, whose value is the id of the session in which this message is sent.

b. The header of each HTTPI message has another new field, called Header Hash. The value of this field is the result of applying the secure hash function SHA-1 on the concatenation of the session key and all the immutable fields in the message header [73].

c. If an HTTPI message has some "content," then the header of the message has the field Content-MD5 [86]. The value of this field is the result of applying the secure hash function MD5 to message content. (Note that this field is optional

in HTTP, but mandatory in HTTPI when the message has some content.)

An illustration of the message exchange between client C and web server S during the session progression phase is shown in Figure 2.1. If HTTPI request is GET request compared to POST request in Figure 2.1, the GET request does not have POST Data and does not need to have H(POST Data) in Header.

The progression phase of an HTTPI connection between client C and server S can last for a long time, for example days or weeks, even if C stops sending HTTPI requests to S. This is in complete contrast with HTTPS connections whose progression phase is usually short, lasting only for minutes, and requires that S continuously sends HTTPS request messages to S. This difference between HTTPI and HTTPS is due to the fact that HTTPI, unlike HTTPS, does not provide any confidentiality guarantees that can be threatened during the long progression phase even when client C leaves the established connection unattended.

2.1.3 Session Termination

At the end of the progression phase of an established HTTPI connection between client C and server S, server S proceeds to terminate the established session. It is also possible that server S may decide to terminate the oldest established HTTPI session in its session table, before the expiration time of this connection, when S notices that the session table has become full or near full.

Server S terminates an established HTTPI session with client C by simply removing the session entry from the session table (in S) and by tearing down the TCP connection, between C and S, over which this HTTPI session was established. Later, if C sends to S a request message in this HTTPI session, C will receive back a TCP reset message informing it that the TCP connection between C and S has been torn down. This will cause C to tear down the TCP connection and remove the session entry associated with its TCP port.

2.1.4 Verifiable Cookies

All the exchanged cookies between a web client C and a web server S in an established HTTPI session are "verifiable cookies." A verifiable cookie is an ordinary cookie with one additional field, called the Verifier. The value of this Verifier field is computed by S as follows:

Verifier := H(Server Key, Session Key, Cookie)

where H is a secure hash function, say SHA-1, applied to the concatenation of three components:

1. Server Key: is a symmetric key that is known only to server S.
2. Session Key: is the session key for the established HTTPI session in which this verifiable cookie is to be sent.
3. Cookie: is the content of the cookie to which this verifier is to be attached.

When server S receives a verifiable cookie (part of an HTTPI request message) that is supposedly sent in some established HTTPI session, then server S examines the verifier field in the cookie and verifies:

a. whether S itself has generated this cookie earlier (since the verifier field is computed using the server key of S, which is known only to S) and
b. whether S itself has sent this cookie earlier in the established HTTPI session (since the verifier field is computed using the session key of the established session)

2.2 Security Guarantees of HTTPI

Our design of HTTPI, which is outlined in the previous Section, provides five security guarantees in the HTTPI communication between a client C and a server S. These five guarantees are as follows:

1. Authentication of server S
2. Integrity of Request Messages from C to S
3. Integrity of Response Messages from S to C
4. Integrity of Cookies from C to S
5. Integrity of Cookies from S to C

In the next five Subsections, we formally state these five guarantees and explain how HTTPI provides them. (Then in the next Section, we discuss how these security guarantees allow HTTPI to defend against some well known cyber attacks.)

2.2.1 Authentication of server S

This security guarantee can be stated as follows:

> If the web client C succeeds in establishing an HTTPI session with some web server claiming to be S, then C is certain that the established HTTPI session is indeed between itself and the web server S.

During the HTTPI session establishment phase, client C and the server execute the following steps (which are part of the TLS-like protocol). First, C receives from the server a signed certificate that has the public key BK of server S. Second, C selects at random part of the HTTPI session key, encrypts the resulting HTTPI session

key SK using BK, and sends the encrypted session key to the server. Third, if the server is indeed server S, then it has the private key of S and can use it to obtain the session key SK from the encrypted session key it has received from C. Fourth, when the server demonstrates to client C that it has succeeded in obtaining the session key SK, client C recognizes that the server is indeed S, and the session establishment phase concludes successfully with both C and S having the same session key SK.

2.2.2 Integrity of Request Messages from C to S

This security guarantee can be stated as follows:

> If the web server S receives an HTTPI request message that is supposedly sent (by some unknown web client) in some established HTTPI session, then S can check whether this message was indeed sent in the established session.

Assume that server S receives an HTTPI request message that is supposedly sent in some established HTTPI session. As S knows the session key SK of the established session, S uses SK to compute the Header Hash of the message. Then, S concludes that the message was indeed sent in the established session iff the computed Header Hash equals the Header Hash field in the header of the received message.

2.2.3 Integrity of response messages from S to C

This security guarantee can be stated as follows:

> If the web client C receives an HTTPI response message that is supposedly sent by some web server S in some established HTTPI session, then C can check whether this message was indeed sent by S in the established session.

Assume that client C receives an HTTPI response message that is supposedly sent in some established HTTPI session. As C knows the session key SK of the established

session, C uses SK to compute the Header Hash of the message. Then, C concludes that the message was indeed sent in the established session iff the computed Header Hash equals the Header Hash field in the header of the received message.

2.2.4 Integrity of Cookies from C to S

This security guarantee can be stated as follows:

> If the web server S receives a verified cookie that is supposedly generated and sent earlier by S in some established HTTPI session, then S can check whether this cookie was indeed generated and sent earlier by S in the established session.

Assume that server S receives a verified cookie that is supposedly generated and sent earlier by S in some established HTTPI session. As S knows its own server key RK and the session key SK of the established session, S uses both RK and SK to compute the Verifier of the cookie. Then, S concludes that the cookie was indeed generated and sent earlier by S in the established session iff the computed Verifier equals the Verifier field in the received cookie.

2.2.5 Integrity of Cookies from S to C

This security guarantee can be stated as follows:

> If the web client C receives a verified cookie that is supposedly sent by the web server S in some established HTTPI session, then C can check whether this cookie was indeed sent by S in the established session.

Assume that client C receives a verified cookie, in the header of an HTTPI response message, that is supposedly sent in some established HTTPI session. From the security guarantee in Section 2.2.3, namely the integrity of response messages from

Table 2.2: Cyber Attacks that HTTPI can defend against

Cyber Attacks	Examples of Attacks	Security Guarantees to Defend Against Attacks
Server Impersonation	Drive-By Pharming [96, 97, 111], DNS rebinding [60], DNS cache poisoning [115]	Server Authentication
Message Modification	In-flight Page Change [98], ARP poisoning [124, 126]	Message Integrity from C to S, Message Integrity from S to C
Cookie Theft	Side Jacking [46], Surf Jacking [43]	Cookie Integrity from C to S
Cookie Injection	Session Fixation [72]	Cookie Integrity from S to C

S to C, C can check whether this message, including the verified cookie in its header, was indeed sent in the established session.

2.3 Defending against Cyber Attacks

If a web client C uses HTTP to communicate with a web server S, then the communication between C and S can be interfered with or disrupted using any of the following four attacks.

1. Server Impersonation
2. Message Modification
3. Cookie Theft
4. Cookie Injection

In this Section, we discuss these four attacks in some detail. We then argue that if client C uses HTTPI (instead of HTTP) to communicate with server S then

none of these attacks can succeed in interfering with or disrupting the communication between C and S.

2.3.1 Server Impersonation

Before a client C can use HTTP to communicate with a server S, C needs first to get the IP address of S. Client C can obtain the IP address of S either from the DNS cache of C itself or through the default DNS server of C. Unfortunately, C may end up obtaining a wrong IP address (rather than the correct IP address of S) as a result of one of the following attacks:

a. DNS Rebinding [60]: This attack corrupts the data stored in the DNS cache of C.

b. DNS Cache Poisoning [115]: This attack corrupts the data stored in the default DNS server of C.

c. Drive-by Pharming [96, 97, 111]: This attack changes the default DNS server of C, from a legitimate DNS server to an adversarial DNS server, without C's knowledge.

When this happens, client C starts to communicate with a different (possibly adversarial) server S' thinking that it is communicating with the intended server S. In effect, the server impersonation attack has succeeded.

Now consider the case where client C uses HTTPI, instead of HTTP, to communicate with server S. Assume that C ends up with a wrong IP address, that belongs to a server S' different from the intended server S. In this case, execution of the TLS protocol between C and S', in the establishment phase of the HTTPI session, will fail. And so the server impersonation attack will not succeed.

2.3.2 Message Modification

If a client C uses HTTP to communicate with a server S, then any computer that is located on the communication path between C and S can modify the exchanged messages between C and S. (Sometimes it is also possible for a computer, that is not on the communication path between C and S, to use ARP poisoning [124, 126] and add itself on the communication path between C and S, and so be able to modify the exchanged messages between C and S.)

There are strong motives for a computer to modify the exchanged messages between a client C and a server S. For example, if this computer belongs to an ISP, then this computer may attempt to insert some advertisements in the web pages that are sent from S to C. On the other hand, if this computer is a proxy of client C, then this computer may attempt to remove all the advertisements from a web page that is sent by S before forwarding the web page to C.

Now consider the case where C uses HTTPI, instead of HTTP, to communicate with S. In this case, if any computer on the communication path between C and S modifies any message that is sent between C and S, then the header hash field and the content-MD5 field in this message will no longer be consistent with the rest of the message and the message ends up being discarded before it is delivered. (Note that the computer, that modified the message, cannot modify the header hash field and the content-MD5 field in the message to make them consistent with the rest of the message. This is because this computer does not know the session key for the current HTTPI session between C and S.)

2.3.3 Cookie Theft

If a client C uses HTTP to communicate with a server S, then all the exchanged cookies in the communication between C and S are sent in the clear (i.e. unencrypted). Thus, any computer C', that is located on the communication path

between C and S, can copy all the cookies that occur in the request messages from C to S and later use these copied cookies to communicate with S pretending to be C. Examples of this attack are Side Jacking [46] and Surf Jacking [43].

Now consider the case where C uses HTTPI, instead of HTTP, to communicate with S. In this case, all the (verifiable) cookies, that are exchanged in the established HTTPI session between C and S, are sent in the clear. Assume that a computer C', located on the communication path between C and S, copies all cookies that occur in the request messages from C to S. (Recall that each one of these cookies has a verifier field whose value is computed as follows:

Verifier := H(Server Key, Session Key, Cookie)

where the session key is the key of the established HTTPI session between C and S.) Now if computer C' later establishes an HTTPI session with server S and pretends to be C by using any of the cookies that it has copied from the former HTTPI session in the later HTTPI session, then the verifier field of this cookie will not be consistent with the session key of the current HTTPI session between C' and S and the cookie will be rejected. (This is because the session key of the later HTTPI session between C' and S is different from the session key of the former HTTPI session between C and S.) In other words, cookies that are stolen from one HTTPI session cannot be used in a later HTTPI session and the cookie theft attack will fail.

2.3.4 Cookie Injection

If a client C uses HTTP to communicate with a server S but ends up, due to some server impersonation attack (similar to those described in Section 2.3.1), communicating with an adversarial server S', rather than S, then S' can send to C erroneous cookies proclaiming that they were sent from server S to client C', rather than C. Unwittingly, client C stores the received erroneous cookies in its cookie jar. Later when client C sends an HTTP request to the true server S, the erroneous cookies

are sent with the request identifying the client to be C', rather than C, and client C is subjected to a cookie injection attack. Examples of cookie injection attacks are reported in [62] and [72].

Note that each cookie injection attack starts with a successful server impersonation attack. But HTTPI can defend against server impersonation attacks, as discussed in Section 2.3.1. Thus, if a client C uses HTTPI to communicate with a server S, then client C cannot be subjected to any cookie injection attack.

For convenience, Table 2.2 lists the cyber attacks that HTTPI can defend against and the security guarantees of HTTPI that can be used to defend against these attacks.

2.3.5 Attacks that HTTPI Cannot Defend Against

HTTPI cannot defend against two types of cyber attacks:

1. *Eavesdropping attacks*:

   Because HTTPI does not provide confidentiality, it cannot defend against eavesdropping attacks. On the other hand, HTTPI is not intended (by design) to provide confidentiality. Thus, applications that value confidentiality should use HTTPS rather than HTTPI.

2. *Attacks that HTTPS cannot defend against*:

   The security guarantees provided by HTTPI is a proper subset of those provided by HTTPS. Thus, attacks that HTTPS cannot defend against, such as XSS attacks [71], CSRF attacks [125], and Phishing attacks [34], HTTPI also cannot defend against.

2.4 Implementation

We explain implementation details of HTTPI in this Section. Hashing header fields and contents seems trivial, but the devils are in detail.

2.4.1 Content Hashing

The Content-MD5 header field is defined as the MD5 hash of an entity-body [86] as follows:

Content-MD5 := H(entity-body)

An entity-body is any content-type data applied with some encoding such as compression [35] as follows:

entity-body := Content-Encoding(Content-Type(data))

If transfer-coding is applied, it becomes a message-body used to carry the entity-body associated with an HTTP request or response [35]. The Content-MD5 header field should be applied to a content after some content encoding, but before some transfer encoding. This definition does not address instance manipulations like range-selection or delta encoding and the concept of instance [84] is introduced. Precisely, the Content-MD5 header field should be applied to a content after some content encoding and before some instance manipulations or some transfer encoding. More precisely, if we consider dynamic contents by server-side scripts, the Content-MD5 header field should be applied to a content after some content encoding and the execution of server-side scripts, and before some instance manipulations or some transfer encoding. Currently, Apache 2.2.11 computes Content-MD5 for static contents and we implemented the filter to compute the Content-MD5 header field for

dynamic contents.

2.4.2 Decoupling Header and Contents

Our first design of HTTPI applies keyed-hashing to entity-header fields and an entity-body together. However, it becomes clear shortly that header fields and content should be decoupled for hashing due to the following two reasons. First, it is inflexible since it still cannot support caching even without encryption, and it has no difference from using TLS without encryption. In fact, TLS supports a null cipher feature such as TLS_RSA_WITH_NULL_SHA or TLS_RSA_WITH_NULL_MD5 [27] though they are not used in practice. Second, it is inefficient since it hurts the pipelining of a web server. The web server can generate the HMAC header field of an instance after reading the instance completely.

Our second design of HTTPI separates header fields from an instance and we use the Content-MD5 header field [86] for contents hashing and keyed-hash header fields with the Content-MD5 header field. It is advantageous in many ways compared to our initial design of HTTPI. First, it is flexible to support caching since the value of contents hashing does not change unless the contents change. Contents hashing can be cached or precomputed if a webpage is static. If the Content-MD5 header field is computed for a static content initially, the Content-MD5 header field can be used for other users. Dynamic webpages can be made possible with two technologies such as client-side scripts and server-side scripts. Since client-side scripts are executed in a browser, the contents hashing needs to consider only server-side scripts for a dynamic webpage. A webpage consists of many web objects including images, stylesheets, and scripts. Contents hashing can be precomputed and cached for images and stylesheets always. Client-side scripts can also be precomputed and cached. Header fields contain more specific information including date, cookies and sometimes authentication. Thus, contents are most likely user-independent and

header fields are user-dependent, and it is reasonable to separate header fields and contents from an instance for hashing. In addition to that, if the Content-MD5 header field is replaced by the Content-SHA1 header field, which does not exist currently, in the future due to the weakness of MD5, the logic of HTTPI need not be changed. Second, it does not impact the pipelining of a web server since it only requires to compute the HMAC of header fields and add the HMAC header field as the last header field on the fly instead of waiting for the computation of keyed-hashing of contents as in our first design.

2.4.3 Our New Header Fields

We design two header fields for HTTPI: 1) HMAC, 2) HMAC-control. We illustrate the HMAC and the HMAC-control header field as follows, respectively. We follow the definitions of Augmented BNF in [35]. *Method*, *Request-URI*, and *Status-Code* of HMAC-control follows the definitions in [35]. The HMAC header field contains the session-id value, the hashing algorithm such as md5 or sha1 and the hash value of header fields with the session key, SK.

HMAC: session-id=quoted-string, alg= md5 | sha1, hash=H(SK,1#header-field)

HMAC-control: http-version=1*DIGIT, method= Method,
request-uri=Request-URI, status=Status-Code,
must-include=*(header-field), must-exclude=*(header-field),
nonce=quoted-string

An HTTP request consists of the request-line, the request header, and the body. Similarly, An HTTP response consists of the status-line, the response header, and the body. The request-line and the status-line can be changed by proxies and should not be used for keyed-hashing directly. Moreover, the request-line and the

status-line processing must be tolerant in a web server and a browser since they can contain extra spaces and tabs [35]. However, the values used by an origin server should be kept since these values can be modified for attacks. The request-line consists of the method, the request-uri, and the http-version, and the status-line consists of the http-version, the status-code, and the reason-phrase. Thus, we add the method, the request-uri, the http-version, and the status-code in the HMAC-control header field. Additionally, we create the *must-include* and the *must-exclude* header fields for the HMAC-control header field. If a 304 (Not Modified) response is used by an origin web server, the cache may include more header fields other than the header fields received by the origin web server. In this case, the origin web server can enumerate all the header fields to compute the HMAC in the *must-include* header field when an HTTP response is received by a browser. If an origin web server is clockless, the origin web server does not generate the Date header field, and proxies may add the Date header field to the header. In this case, the origin web server can use the *must-exclude* header field to note that the origin web server does not generate the Date header field.

2.4.4 Caching in HTTPI

In order to support the caching mechanism in HTTP, HTTPI is required to keyed-hash header fields selectively. There are two kinds of header fields depending on the behavior of caching: end-to-end header fields and hop-by-hop header fields. The following HTTP/1.1 header fields are hop-by-hop headers: Connection, Keep-Alive, Proxy-Authenticate, Proxy-Authorization, TE, Trailers, Transfer-Encoding and Upgrade [35]. All the other header fields defined by HTTP/1.1 are end-to-end header fields. End-to-end header fields should be included for computing the HMAC header field, but hop-by-hop header fields should be excluded. Other hop-by-hop header fields must be listed in the Connection header field to be introduced into

HTTP/1.1 or later [35] and these header fields should be excluded, too. We found that Via and Warning header fields can be modified in-transit and they should be excluded for computing the HMAC header field.

Some proxies might convert original contents to some other new formats and can break the Content-MD5 header field. There are two types of proxies such as a transparent proxy and a non-transparent proxy. A transparent proxy passes requests and responses unmodified whereas a non-transparent proxy modifies requests and responses to converts between image formats for saving cache space or reducing the amount of traffic. Unfortunately, if a non-transparent proxy converts original contents to some other new formats, HTTPI cannot work since the Content-MD5 header field will be different for a new format. If an HTTP message includes the *no-transform* directive, the cache or the proxy should not change any aspect of the entity-body specified by the Content-Encoding, the Content-Range, and the Content-Type header field including the entity-body itself [35]. Thus, *no-transform* should be used with HTTPI.

When a cache makes a request to an origin web server, and the origin web server provides a 304 (Not Modified) response or a 206 (Partial Content) response, the cache then constructs a response and send the response to a browser. The 304 response from the origin web server contains only header fields and the cache retrieves the entity-body stored in the cache entry and combine the header fields and the entity-body to construct an HTTP response to the browser. The origin web server can still use HTTPI for this caching protocol if the origin web server includes the Content-MD5, the HMAC and the HMAC-control header field. Since the HMAC header field is computed only with header fields, the origin web server can compute the HMAC header field and the origin web server needs to enumerate all the header fields to compute the HMAC header field in the *must-include* header field since the cache may include more header fields than the header fields provided

by the origin web server. When the cache receives the 304 response from the origin web server, the cache can combine the entity-body in the cache entry as usual. When the browser receives the HTTP response from the cache, the browser can check the HMAC header field by computing all the enumerated header fields in the *must-include* header field and the Content-MD5 of the entity-body. The request to the 304 response includes the If-Modified-Since or the If-None-Match header field to check whether objects are modified or not after the browser receives the content previously. The If-Modified-Since header field is based on the Last-Modified header field and check whether the objects are modified from the date in the Last-Modified header field. The If-None-Match header field depends on the ETag header field and the origin web server should ensure that the ETag header field is uniquely changed whenever a content is changed. In both cases, the origin web server might not be able to generate the Last-Modified or the ETag header field if web objects are generated dynamically from a database. It is difficult to know when the objects are generated and how the objects have a unique ETag if they are generated dynamically. Due to these limitations, Nottingham proposes to use the Content-MD5 header filed for a strong cache validation with a new header field called *If-Not-Hash* instead of the If-Modified-Since and the If-None-Match [91] header field. Moreover, MD5 hash can be used to detect duplicate transfer [85]. A traditional web cache indexes each entry by a given URL, but this can cause a redundant payload transfer by a cache miss between proxies and origin web servers. Therefore, Content-MD5 can be beneficial not only for integrity but also for performance.

2.5 Compatibility with Middle Boxes

We argued in the previous two Sections that HTTPI has several security advantages over HTTP: first HTTPI provides some security guarantees that cannot be provided by HTTP (Section 2.2), and second HTTPI can defend against some cyber attacks

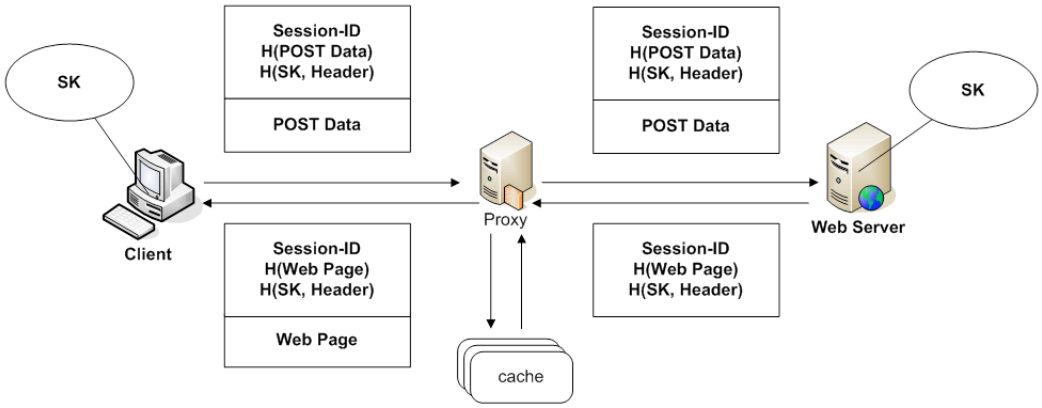


Figure 2.2: Compatibility with Cache Proxies

that cannot be defended against by HTTP (Section 2.3). This means that the World Wide Web needs to support HTTPI beside, or instead of, HTTP especially since we show in the next Section that the performance of HTTPI is very close to that of HTTP.

On the other hand, one may argue that the web may not need to support HTTPI beside HTTPS since HTTPS can be used, instead of HTTPI, in those applications that only need HTTPI. But we refute this argument in two ways. First, we argue in this Section that HTTPI is compatible with middle boxes, such as cache proxies and application firewalls, in the Internet whereas HTTPS is not. Second, we show in the next Section experimental results which demonstrate that the throughput of HTTPI is 37% higher than that of HTTPS and the CPU execution time of HTTPI is 23% lower than that of HTTPS. Therefore, there are significant performance gains to be had when HTTPI is used in place of HTTPS in those applications that do not require message confidentiality.

In the remainder of this Section, we discuss how HTTPI is compatible with two important types of middle boxes: cache proxies and application firewalls.

2.5.1 Compatibility with Cache Proxies

Consider the case where a client C uses HTTPI to communicate with a web server S, and assume that all the exchanged (request and response) messages between C and S reach a cache proxy PR after they are sent and before they reach their ultimate destinations. Note that PR does not know the session key for the current HTTPI session between C and S. Yet, PR can still read each request message from C to S (since none of the messages is encrypted) and determine whether or not the requested web page in the request message is already stored in PR.

There are two possible scenarios in this case:

1. If the requested web page is not in PR, then PR forwards the request message to S. Later, when S sends back the requested page in a response message, PR stores a copy of the requested page in its memory before forwarding the response message to C.

2. If the requested web page is already in PR, then PR applies the secure hash function MD5 to the page and sends the result along with the session ID (for the HTTPI session between C and S) to server S. Later server S computes the header hash for the requested page and sends it to PR. Then PR prepares the response message, that has the requested web page and the header hash, and sends it to C. The exchanged messages in this scenario are illustrated in Figure 2.2.

Note that in Scenario 1, the requested web page was sent all the way from S to PR then to C, whereas in Scenario 2, the requested page is sent only from PR to C. Thus, the saving in communication is achieved since in most cases Scenario 2 is much more likely to occur than Scenario 1.

2.5.2 Compatibility with Application Firewalls

Consider the case where a client C uses HTTPI to communicate with a web server S. Assume that all the request messages from C reach a server firewall SF before reaching S, and all the response messages from S reach a client firewall CF before reaching C.

When SF receives a request message, it checks whether the cookies in the message header are correct (i.e. could have been sent earlier by server F), and whether the Javascript code in the POST data of the message, if any, is harmless. Based on these checks, SF decides either to forward the request message to server S or to discard the message.

When CF receives a response message, it checks whether the Javascript code in the web page in the message, if any, is harmless. Based on these checks, CF decides either to forward the response message to client C or to discard the message.

Note that the two application firewalls SF and CF can perform their functions, even though they do not know the session key for the current HTTPI session between C and S, because none of the exchanged messages between C and S is encrypted.

2.6 Experimental Results

In this Section, we describe an experiment that we carried out to compare the performance of HTTPI against the performance of HTTP and of HTTPS, when HTTP and HTTPS are used in place of HTTPI.

This experiment involves a client machine and a server machine with Ubuntu 8.04. The client machine is an Intel Core 2 Duo CPU @ 3.16 GHz with 2 GB RAM. The server machine is an Intel Core 2 Duo CPU @ 3.00 GHz with 2 GB RAM. The client and the server machine are connected using a 100 Mb/second Ethernet. The

server machine hosts an Apache version 2.2.11 server which supports both HTTP and HTTPS. We augmented this Apache server with a new module that implements HTTPI.

We made the augmented Apache server host three web pages, which we obtained from the web: an Amazon page, a Facebook page, and a New York Times page. The characteristics of these three pages are as follows:

(1) The Amazon page consists of a container HTML page that has 172 KB, and 53 files of images, scripts, and style sheets totaling 484 KB.

(2) The Facebook page consists of a container HTML page that has 360 KB, and 51 files of images, scripts, and style sheets totaling 1.1 MB.

(3) The New York Times page consists of a container HTML page that has 140 KB, and 94 files of images, scripts, and style sheets totaling 1.1 MB.

The experiment consists of three stages:

(1) In the first stage, the client machine communicates with the server machine using HTTPI and the two machines execute X transactions, where X = 20, 60, and 100 and a transaction consists of the client machine sending one request message and the server machine replying back with a response message that includes the requested web page (Note that the value of X is chosen to be relatively large since the lifetime of an HTTPI connection is intended to be relatively long as discussed in Section 2.1.2 above).

(2) The second stage of the experiment is the same as the first stage except that the client machine and the server machine communicate using HTTP (instead of HTTPI).

(3) The third stage of the experiment is the same as the first stage except that the

client machine and the server machine communicate using HTTPS (instead of HTTPI).

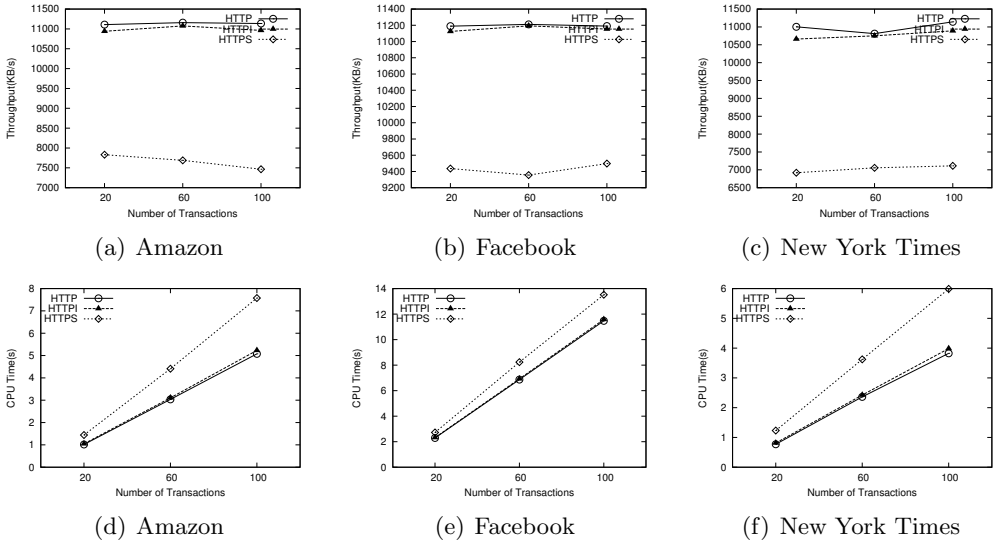
In each stage of the experiment we measured two parameters: network throughput (in KB/second) and CPU execution time (in seconds). We ran each stage 5 times and measured the average. The measured results of this experiment are shown in Figure 2.3(a) to 2.3(f). Figure 2.3(a), 2.3(b), and 2.3(c) show the network throughput when the requested web page is Amazon, Facebook, and New York Times, respectively. Figure 2.3(d), 2.3(e), and 2.3(f) show the CPU execution time when the requested web page is Amazon, Facebook, and New York Times, respectively. From these figures, we conclude that the throughput of HTTPI is within 1.2% from that of HTTP and is about 37% better than that of HTTPS. We also conclude that the CPU time of HTTPI is within 1.9% from that of HTTP and is about 23% better than that of HTTPS.

From these results, we conclude that there is a significant performance gain that can be achieved by using HTTPI, instead of HTTPS, when message confidentiality is not required. Therefore, supporting HTTPI beside HTTPS in the web seems to be a reasonable design option.

For completeness, we repeated our experiment when the client machine and the server machine are connected using a 1 Gb/second fast Ethernet in Figure 2.4. The results show the same trends as those in Figure 2.3.

2.7 Related Work

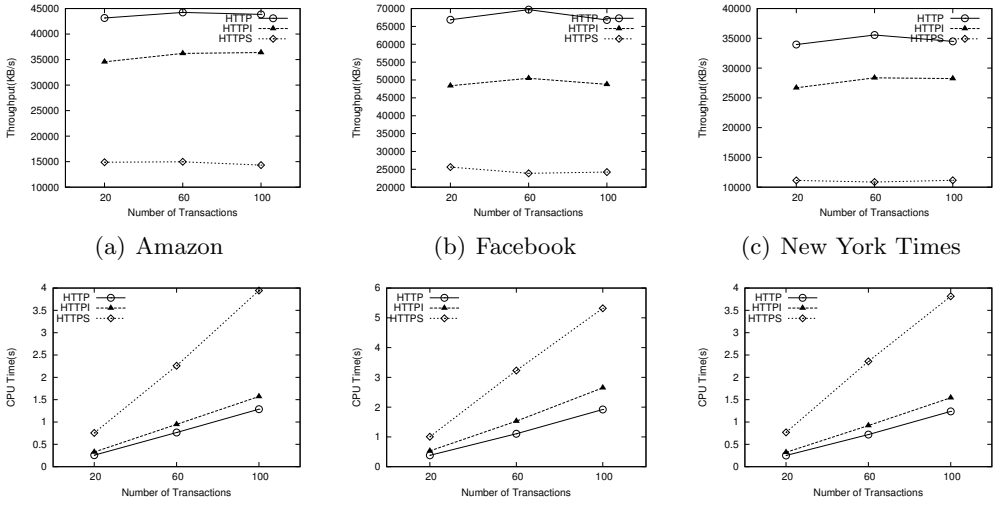
Many solutions for the message modification problem of HTTP page are proposed, but none of these address server impersonation, message integrity, and cookie theft and injection attacks comprehensively like HTTPI. A solution for server impersonation attacks will not work as one for message modification, and cookie theft.



(a) Amazon (b) Facebook (c) New York Times

(d) Amazon (e) Facebook (f) New York Times

Figure 2.3: Performance Comparisons between HTTP, HTTPS, and HTTPI in 100 Mbps Internet



(a) Amazon (b) Facebook (c) New York Times

(d) Amazon (e) Facebook (f) New York Times

Figure 2.4: Performance Comparisons between HTTP, HTTPS, and HTTPI in 1 Gbps Internet

Likewise a solution for DNS rebinding will not work as one for DNS cache poisoning though these attacks are server impersonation attacks. On the other hand,

HTTPI defends against these attacks totally. Furthermore, an attacker can create new attacks by sidestepping existing defenses and combining attacks vectors like server impersonation, message integrity, and cookie theft and injection. Nevertheless, HTTPI defends against not only existing attacks, but also these new attacks.

2.7.1 Server Authentication

Attackers launch server impersonation attacks in many ways such as Pharming [96, 97, 111], DNS rebinding [60], DNS cache poisoning [115], and ARP poisoning [124, 126]. These attacks are possible because identities are not correctly bound to its indirection. For exaple, attackers need to map domain names with attackers' IP addresses for DNS attacks. Similarly, attackers need to associate IP addresses with attackers' MAC addresses for MAC attacks.

A solution for DNS rebinding does not work as one for DNS cache poisoning. For example, DNS pinning is a classic defense against DNS rebinding. DNS pinning keeps the DNS mappings in a browser cache for a certain period of time and prevents a domain name from being rebound with an attacker's IP address. DNS pinning works for DNS rebinding, but it does not address DNS cache poisoning. DNS cache poisoning corrupts the DNS mapping in a default DNS's cache, but DNS rebinding corrupts the DNS mapping in a browser's cache.

A solution for DNS misbinding, DNS rebinding, does not work as one for MAC misbinding. DNS pinning concerns the mapping between IP addresses and domain names, and MAC misbinding concerns the mapping between IP addresses and MAC addresses. For example, SSLStrip [76] is a Man-in-the-Middle (MITM) attack using ARP poisoning. Most HTTPS-enabled web sites allow a user to initiate an HTTPS session with HTTP. If the user types a URL without `https`, the user's browser assumes its protocol as `http`. Then, a target web site redirects the user to `https`. SSLStrip makes use of this convention and an attacker uses ARP poisoning

to route packets to the attacker's MAC address. When the user initiates `http` connection with the target web site, the attacker establishes an HTTP connection with the browser and an HTTPS connection with the target web site. The HTTP-to-HTTPS redirection problem is addressed by ForceHTTPS [59]. If ForceHTTPS cookie is set or configured by a user, the user's browser initiates a protocol with `https` instead of `http` by URL rewrite rules.

In order to resolve server impersonation attacks, we must take every binding into account. Instead, we might consider the root cause of server impersonation attacks and ensure server authentication for misbinding. HTTPI is not designed to solve any specific server impersonation attack in mind, but HTTPI addresses the set of problems, server impersonation attacks with server authentication. Thus, HTTPI provides a consistent defense against server impersonation attacks.

2.7.2 Message Integrity

HTTP provides the Content-Length [35] and the Content-MD5 [86] header fields for message integrity. The Content-Length header field provides the size of the content in a web page by bytes. The Content-MD5 [86] header field indicates the MD5 hash of the content in a web page. Given the Content-MD5, it is hard to find the same MD5 value with a different content. On the contrary, given the Content-Length header field, it is easy to modify the content by substitution. But, in both cases, there is no guarantee that those header fields are neither modified nor omitted.

Web Tripwire [98] is an integrity mechanism to detect the modification of HTTP pages in web applications by comparing requested HTTP pages with the known good representations of requested HTTP pages using a tripwire script. The detection mechanism of Web Tripwire is limited to received contents, especially only to an HTML page and there is no page integrity from a client to a web server. In addition to that, there is no message integrity about HTTP redirection or HTTP

error messages though these redirections or error messages are easily used by attackers to trick users. Furthermore, Web Tripwire requires more bandwidth for the known good representations of any requested web page by 17%.

HTTPI requires three more header fields of 160 bytes and the size of these header fields does not vary depending on the size of contents, but Web Tripwire requires more bandwidth depending on the size of contents. Nevertheless, Web Tripwire is not cryptographically secure and a false positive and a false negative can occur. On the other hand, HTTPI provides a complete solution for message integrity in HTTP pages without false positives and false negatives. HTTPI provides message integrity for both HTTP requests and HTTP responses and for any web objects. Saltzman and Sharabani proposes HTTP Response Signing [104], but signing requires more computation than hashing and it only protects HTTP responses.

HTTP header fields can be omitted or modified, but new HTTP header fields are proposed to defend against web attacks. Without HTTP header integrity, these new HTTP header fields are futile. For example, the HTTP Referer header field can be used to mitigate CSRF attacks. Furthermore, the HTTP Origin header field [11] is proposed for Login CSRF attacks due to the privacy leaks by the HTTP Referer header field. Yet, HTTP does not have any proper header protection. HTTPI provides message integrity for HTTP header fields. Thus, HTTPI can be complementary to many proposals relying on HTTP header fields.

2.7.3 Cookie Integrity

SessionLock [2] is proposed for cookie theft. SessionLock secures web sessions against SideJacking [46] by using a session secret shared between a browser and a web server over TLS. The browser uses the session secret to authenticate to the web server by keyed-hashing timestamp and a request URL in every subsequent HTTP request. Fu et al. [40] propose to use a server key to protect cookies from being forged, but

the cookies using a server key can be replayed until the cookies expire. HTTPI defends against cookie theft with the cookie verifier. The cookie verifier protects cookies from being forged and replayed. If only a server key is used, it protects cookies from being forged, but not replayed. HTTPI protects cookies from being replayed as well as forged by using a session key and a server key.

Session Fixation [72] is a cookie injection attack. Attackers can inject cookies by server impersonation or by message modification. With HTTPI, attackers cannot inject cookies since HTTP provide server authentication and message integrity. Even though attackers inject cookies, web servers can verify and reject cookies with the cookie verifier.

2.8 Concluding Remarks

The research reported in this Chapter is motivated by the observation that the transport protocol HTTPS, which provides security over the World Wide Web, is overkill for many open web applications, such as web email, social networking, and web blogging. This is because HTTPS provides message confidentiality to all web applications, typically at a high cost, even when the web applications do not require message confidentiality.

The cost of HTTPS providing message confidentiality is two-fold:

1. Because HTTPS encrypts and decrypts each sent (request or response) message, the throughput of HTTPS is reduced by about 40%.

2. Because each HTTPS (request or response) message is encrypted using a symmetric key that is not known to any of the middle boxes in the Internet, none of these middle boxes can process these messages. Thus, all the gains that could have been offered by the middle boxes in the Internet are lost whenever HTTPS is used.

To resolve this mismatch between what HTTPS provides and what many open web applications require, we propose in this Chapter a new web transport protocol named HTTPI. This new protocol is designed to provide all the security guarantees provided by HTTPS, except one – message confidentiality. Thus, web applications that do not require message confidentiality, and there are many of them, can be deployed on top of this new protocol rather than on top of HTTPS. (However, web applications, that do require message confidentiality, such as web banking, can still be deployed on top of HTTPS.)

There are two significant advantages of using HTTPI over using HTTPS:

a. Our experimental results, in Section 2.6, showed that the throughput of HTTPI is almost 40% better than that of HTTPS.

b. As discussed in Section 2.5, HTTPI is compatible with, and can take full advantage of, the middle boxes in the Internet. By contrast, HTTPS is not compatible with, and cannot utilize any of the middle boxes in the Internet.

Our experimental results in Section 2.6 also showed that the throughput of HTTPI is within 1.2% of that of HTTP. Therefore, if HTTPI happens to replace HTTP as the baseline transport protocol over the web, then the reduction in throughput can go unnoticed by most web users. On the other hand, the improvement of security (e.g. as discussed in Section 2.3, HTTPI can defend against Pharming attacks but HTTP cannot) can be appreciated and cheered by all users.

Finally, we observe that the relationship between HTTPI and HTTPS is analogous to the relationship between the IP Authentication Header (AH) [68] and the IP Encapsulation Security Payload (ESP) [69] in IPsec [70]. Thus, just as both AH and ESP are supported by IPsec, both HTTPI and HTTPS should be supported by the web.

Chapter 3

TLP: Transport Login Protocol

The conventional wisdom has always been that users should refrain from entering their sensitive data (such as usernames, passwords, and credit card numbers) into HTTP (or white) pages, but they can enter these data into HTTPS (or yellow) pages. Unfortunately, this assumption is not valid as it became clear recently that, through human mistakes or Phishing or Pharming attacks, a displayed yellow page may not be the same one that the user had intended to request in the first place. In this Chapter, we propose to add a third class of secure web pages called brown pages. We show that brown pages are more secure than yellow pages especially in the face of human mistakes and Phishing and Pharming attacks. Thus, users can enter their sensitive data into brown pages without worry. We present a login protocol, called the Transport Login Protocol or TLP for short. An HTTPS web page that is displayed on the browser is classified brown by the browser if and only if this web page has been called into the browser either through TLP or from within another brown page that had been called earlier into the browser through TLP. TLP was published in [22].

When a user needs to display a web page on his browser, the user follows any one of four direction rules, described below, to request that his browser calls

the page and displays it on the screen. If the requested page is an (insecure) HTTP page, then the browser calls the page and displays it without any firm guarantee that the displayed page is the one that the user has requested. On the other hand, if the requested page is a (secure) HTTPS page, then the browser displays the page only after it has authenticated that the page is the one that the user has requested. Unfortunately, as described below, the authentication procedure is vulnerable to human mistakes, by the user, and to Phishing and Pharming attacks [92], by adversarial web sites. And so it is possible that the displayed page may not be the one requested by the user after all.

The user may not mind that the displayed page is different from the page that he has requested for two reasons. First, both the displayed page and the page that the user has requested have similar graphics and colors and the user may not notice that the displayed page is actually not the one that he has requested even in the presence of security indicators [26]. Second, the user may notice that the displayed page is not the one that he has requested, but he may believe that the displayed page is a legitimate redirection that was requested by the page that he has requested. In any case, the user may proceed to enter some sensitive data, such as his credit card number, into the displayed page which may happen to be an adversarial page.

This Chapter is dedicated to prevent these scenarios from occurring. Towards this end, we propose to introduce a new class of HTTPS web pages, which we refer to as brown pages. As discussed below, brown pages are secure against human mistakes and Phishing and Pharming attacks. Thus, when a user requests that his browser calls and displays an HTTPS page and then the browser displays the page and classifies it brown, the user knows that the displayed page is indeed the one that he has requested and so he can proceed to enter his sensitive data into it.

In order for the browser to be able to classify a called HTTPS page brown,

the browser needs to call this page through a login protocol that is completely secure against human mistakes and Phishing and Pharming attacks. In this Chapter, we present and discuss the design and implementation of such a login protocol.

3.1 Attack Scenarios

In this Section, we describe three attack scenarios, caused by human mistakes or Phishing or Pharming attacks [48,78]. In each one of these scenarios, a user intends to call into his browser a particular HTTPS page, but he ends up calling a wrong HTTPS page into his browser.

1. **Human Mistakes**:

   A user intends to enter the URL `https://www.amazon.com` into the URL box of his browser. But he enters the wrong URL `https://www.anazon.com` by mistake.

2. **Phishing Attacks**:

   A user receives an email that urges the user to click on a link described as leading to the web site `https://www.amazon.com`. By clicking on this link, the user ends up in the wrong web site `https://www.anazon.com`.

3. **Pharming Attacks**:

   For the convenience of its users, the web site `https://www.amazon.com` allows its users to call the web site using the alternative insecure URL `http://www.amazon.com`. Now, the DNS of a user can be manipulated so that when the user uses this insecure URL to request the web site, the user's DNS directs the request to an adversarial web site that redirects the user's browser to the wrong web site `https://www.anazon.com`.

In each one of these three scenarios, the user intended to call into his browser the web site `https://www.amazon.com`, but he ends up calling the wrong web site `https://www.anazon.com`. The user does not notice the switch, from `https://www.amazon.com` to `https://www.anazon.com`, because the two web sites have similar logos, graphics, and colors, and maybe similar URLs. Thus the user proceeds to enter his sensitive information (such as username, password, or credit card numbers) into the wrong web site. The objective of this Chapter is to outline a proposal to counter these three attack scenarios.

One method to counter these scenarios is to advise the user to be careful and check the URL box of the displayed HTTPS web page on his browser before he enters his sensitive data into the displayed web page. However, it is very difficult for a user to remember and follow this advice every time he requests an HTTPS web page.

A second method to counter these scenarios is to make the browser check, before it displays an HTTPS web page, that this page is indeed the one that the user wants. Unfortunately, the browser cannot tell whether or not the user wants the web page whose URL is in the URL box.

The method that we adopt in this Chapter to counter these scenarios is as follows. Browser B of user U displays an HTTPS page from a web site S when and only when the following three conditions hold.

1. User U has requested the page.

2. Site S has verified that sometime in the past user U has registered and stored his login data in site S.

3. Browser B has verified that sometime in the past user U has registered and stored his login data in site S.

If any one of these three conditions does not hold, then the browser of user U refuses to display the requested page. The correctness of this method is based on the reasonable assumption that each web site in which user U registers is a legitimate, rather than an adversarial, site. Next, we argue that this method can counter the above three scenarios.

Consider the first scenario. If user U intends to request the web site `https://www.amazon.com`, but by mistake requests the web site `https://www.anazon.com`, then only one of two outcomes is possible. The most likely outcome is that user U has not registered in the web site `https://www.anazon.com` and so the browser of user U will not display the web page. The second outcome is that user U has registered in the web site `https://www.anazon.com` and so the browser will display the legitimate web page of this site and user U will notice that the displayed page is not the one that he wants.

Now consider the second and third scenarios. In these scenarios, the web site `https://www.anazon.com` is an adversarial site and so user U has not registered in it. Thus, the browser of user U will not display the web page.

3.2 Countering the Attack Scenarios

In this Section, we outline our proposal to modify the browser and some web sites in order to counter the attack scenarios, caused by human mistakes and Phishing and Pharming attacks, discussed in the previous Section. Our proposal consists of three parts.

1. **White, Yellow, and Brown Pages**:

   We propose to modify the browser so that the browser classifies each displayed HTTP web page as white, and classifies each displayed HTTPS web page as either yellow or brown. As described below, a user should regard each white

page as insecure, each yellow page as mildly secure (which means that the page is vulnerable to human mistakes and Phishing and Pharming attacks), and each brown page as highly secure (which means that the page is secure against human mistakes and Phishing and Pharming attacks).

2. **A New Login Protocol**:

   We also propose to add a new login protocol to the browser and to some web sites that need to be (extra) secure against human mistakes and Phishing and Pharming attacks. We call this new login protocol the Transport Login Protocol or TLP for short. When a user invokes TLP on his browser and requests the browser to call a web page on a specified web site, the following three steps are executed. First, the browser and the specified web site use TLP to establish mutual authentication between each other. Second, if the mutual authentication between the browser and the web site succeeds, then the web site redirects the browser to an HTTPS web page. Third, the browser calls the secure web page and, upon receiving it, the browser assigns it a brown classification and displays it to the user.

3. **Classification of Web Pages**:

   The modified browser assigns a classification, white, yellow, or brown, to each displayed web page, depending on how this page has been called into the browser in the first place. Thus the same displayed HTTPS page can be assigned a yellow classification if it is called into the browser one way, and assigned a brown classification if it is called into the browser another way. We adopt the following classification rules.

   (a) Any HTTP page, that is called into the browser, is classified white by

the browser.

(b) Any HTTPS page, that is called into the browser using our login protocol TLP, is classified brown by the browser.

(c) Any HTTPS page, that is called into the browser using the TLS protocol [14], is either classified yellow if this page is called from within a displayed white or yellow page, or classified brown if this page is called from within a displayed brown page.

When the browser displays a web page, the browser makes its classification of the displayed page clear to the user by choosing an appropriate background color for the URL box. If the displayed page is white (or yellow or brown respectively), then the background color for the URL box is white (or yellow or brown respectively). Note that the current browser already supports white and yellow classifications of web pages. So the main contributions of this project are merely the addition of brown classifications and the introduction of the new login protocol TLP which can be used in calling brown web pages into the browser.

(Recently, a green classification of HTTPS web pages has been introduced to distinguish those HTTPS pages that have extended validation certificates [36]. Clearly some green pages, like yellow pages, can still be adversarial, and can still be used in launching Phishing and Pharming attacks as described above. Henceforth, when we refer to yellow pages, we do mean yellow or green pages.)

The policy for entering sensitive data (such as usernames, passwords, and credit card numbers) into a displayed web page depends on the classification of the displayed page. This policy consists of the following three rules.

1. **The White Page Rule**:

   A user should never enter sensitive data into a white page.

2. **The Brown Page Rule**:

   A user can enter sensitive data into a brown page.

3. **The Yellow Page Rule**:

   Before a user can enter sensitive data into a yellow page, the user should have prior knowledge that this data can be entered into this particular page, and the user should check that the URL box of the displayed page has indeed the URL of this particular page.

3.3 The Current Login Protocol

Our login protocol TLP, described in the next Section, enjoys a number of nice features that are not all present in any of the current login protocols. These nice features are as follows.

1. **Immunity to Attacks**:

   When a user U uses TLP to log into a site S, then the login succeeds if and only if both browser B of user U and site S can verify that user U has registered (and stored some login data) in site S sometime in the past.

2. **No External Servers**:

   All the login data, that are needed by user U to use TLP and successfully log into site S, are stored on site S. Thus TLP does not need any external servers to store some of the login data.

3. **One-Time Login Data**:

   In TLP, the login data, that are needed by user U to log into site S, are

updated after each successful login of U into S. Therefore, if an adversary somehow steals the login data of user U in site S, then the stolen data becomes useless after the next login of U into S.

4. **Universal Passwords**:
   Each user U needs only to memorize one password P, called the TLP universal password of U. User U employs his universal password in the TLP protocol to log into every web site. No web site S can deduce the TLP universal password of user U from the login data that user U stores in S or from the messages exchanged between the browser of U and site S during the execution of TLP.

5. **Standard Cryptography**:
   TLP uses only standard symmetric cryptography and standard secure hash functions. Thus, every time the standards of symmetric cryptography or of hash functions are updated, the standards of TLP are updated accordingly.

Next we argue that none of the login protocols, that have been proposed recently, enjoys all these five features.

The current login protocol over the web consists of two protocols: the standard TLS protocol [14] (which is used to authenticate a secure web site by the browser), and a non-standard password protocol (which is sometimes used to authenticate the secure web site by the user and to authenticate the user by the secure web site). As described in Section 3.1, this login protocol is vulnerable to human mistakes and Phishing and Pharming attacks, and so it does not enjoy feature 1.

This login protocol can be strengthened using Site Keys which allow a user to authenticate the identity of the web site being logged into [10, 108]. Unfortunately, Site Keys can be stolen using Man-In-The-Middle Attacks. Thus the strengthened

protocol still does not enjoy feature 1.

The login protocol SRP [119, 120] does not enjoy any of features 3, 4, and 5 above.

The hash-based protocols, such as [41, 45, 49, 67], enjoy the features 2, 4, and 5. They also allow the web site to verify that the user has registered in the site sometimes in the past. Unfortunately, they do no allow the user's browser to verify that the user has registered in the site sometime in the past. Thus these protocols do not enjoy feature 1. Also, some of these protocols, for example [49], do not enjoy feature 2.

The Passpet system [122] does not enjoy features 2 and 3.

3.4 The New Login Protocol

Our login protocol TLP is to be executed between browser B of user U and web site S. Prior to executing TLP, user U needs to have registered with site S by making its browser B store in S the following tuple of four data items:

$$(H(U), \quad n, \quad H(0, n, P, S), \quad H^2(1, n, P, S))$$

where

U	is the username of the user,
B	is the browser of user U,
n	is a nonce selected at random by browser B,
H	is a standard secure hash function,
0	is the character zero,
1	is the character one,
P	is the TLP universal password of user U, and
S	is the domain name of the web site.

Note that $H(0, n, P, S)$ denotes the application of the secure hash function H to the concatenation of the four data items 0, n, P, and S. Also, $H^2(1, n, P, S)$ denotes two consecutive applications of function H to the concatenation of the four data items 1, n, P, and S. After B stores this tuple in S, B forgets the tuple completely.

Executing TLP between browser B and site S is intended to achieve five objectives.

1. B checks that S is one of the sites where user U had previously registered (and stored the above tuple of four data items).

2. S checks that user U has entered his universal password P to browser B.

3. Both B and S agree on a symmetric session key that they can use to encrypt and decrypt their exchanged messages.

4. B selects a new random nonce n' and stores the following tuple of four data items in S in place of the above tuple:

$$(H(U), \quad n', \quad H(0, n', P, S), \quad H^2(1, n', P, S))$$

   (Therefore, each successful login of browser B into site S causes the tuple of four data items that B had previously stored in S to be replaced by a new tuple of four data items also provided by B.)

5. S sends to B the URL of the next HTTPS page that B needs to call, using TLS, along with a cookie identifying user U and testifying that the login procedure between U's browser and S, has been successful. When the next HTTPS page is called into B, B assigns this page a brown classification. Moreover browser B assigns any other HTTPS page, that is called using TLS from within this brown page, a brown classification .

We adopt the following notation in describing a field in a message that is sent during the execution of TLP.

$$[expression1] < expression2 >$$

This notation means that the value of *expression1* is used as a symmetric key to encrypt the value of *expression2* before the message is sent.

To start executing TLP between B and S, user U enters three data items, namely U, P, and S, to a local web page named `httpl` stored in browser B. Then the execution of TLP proceeds with the following four message exchanges between B and S.

$B \rightarrow S:$ {Hello Message}
U

$B \leftarrow S:$ {Hello-Reply Message}
$n, \quad [H(0, n, P, S)] < SN >$

$B \rightarrow S:$ {Login Message}
$U,$
$[H^2(1, n, P, S)] < H(1, n, P, S), BN, H^2(1, n', P, S) >,$
$[H(BN, SN)] < n', H(0, n', P, S) >$

$B \leftarrow S:$ {Login-Reply Message}
$[H(BN, SN)] <$ URL of next HTTPS web page $>,$
$[H(BN, SN)] <$ cookie $>$

The hello message, from B to S, consists of the username of user U who wants to log into site S. On receiving this message, S fetches the tuple

$$(H(U), n, H(0, n, P, S), H^2(1, n, P, S))$$

that B had stored previously in S. Then S uses the data item $H(0, n, P, S)$ as a symmetric key to encrypt a new nonce SN that S selects at random. The result of the encryption is denoted $[H(0, n, P, S)] < SN >$ and is included in the hello-reply message that is sent from S to B.

After B receives the hello-reply message, it computes $H(0, n, P, S)$ and uses it to obtain the nonce SN from the received message. Then, B selects at random two nonces BN and n', and uses the received SN and the computed BN and n' to construct the login message before sending it to site S.

After S receives the login message, it performs four tasks. First, it checks that user U has indeed entered its TLP universal password P into browser B. Second, S extracts the nonce BN from the received message, and now both B and S know BN and SN. Third, S stores the tuple:
$(H(U), \quad n', \quad H(0, n', P, S), \quad H^2(1, n', P, S))$ in place of the earlier tuple. Fourth, S constructs the login-reply message and sends it to browser B.

After B receives the login-reply message, it concludes that S is one of the web sites where user U has previously registered. Moreover, B gets the URL of the HTTPS page that B needs to call next using TLS, along with a cookie that identifies user U and testifies to the fact that the login procedure between U's browser and S has been successful.

Figure 3.1 illustrates the five steps that are needed for a user to use TLP to log into a web site in a domain say `xyz.com`.

1. The user calls a local web page, for convenience named the `httpl` page, on his browser and enters his username, his TLP universal password, and the site address `www.xyz.com` into this page.

2. The browser uses DNS to get the IP address of site `www.xyz.com`.

3. The browser and site `www.xyz.com` execute TLP. At the end, the browser

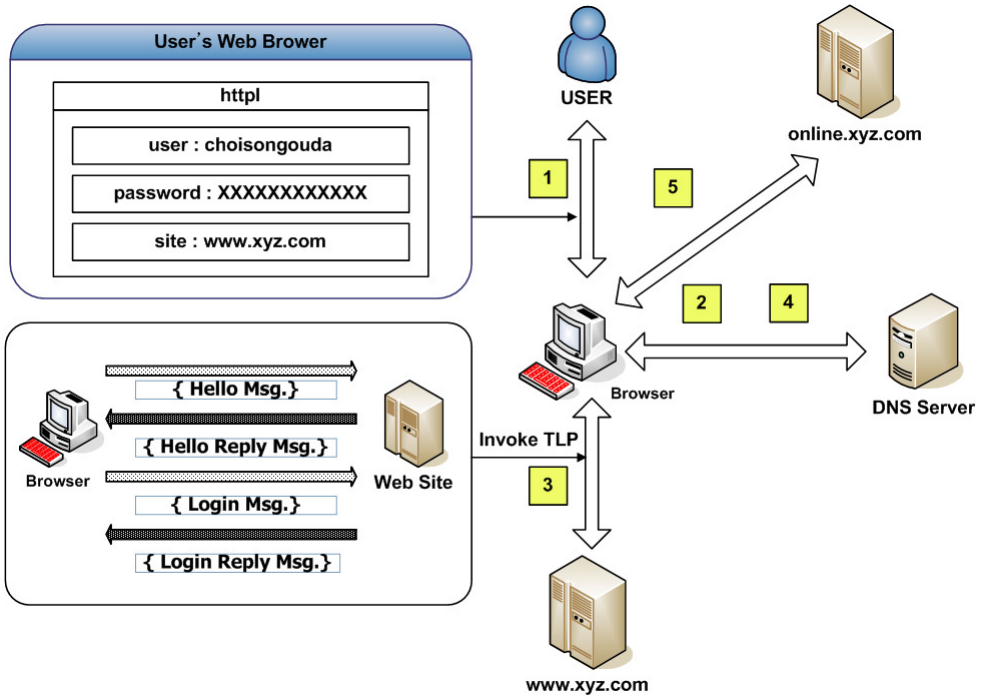


Figure 3.1: Using TLP

receives the URL of a web page on site `online.xyz.com` and a cookie.

4. The browser uses DNS to get the IP address of site `online.xyz.com`.

5. The browser and site `online.xyz.com` execute TLS, and the browser gets the required HTTPS page at the end. The browser classifies this page brown. It also classifies any other HTTPS page, that is called using TLS from within this page, brown.

3.5 Correctness of TLP

An adversary has no chance to succeed in attacking TLP between the browser of user U and web site S unless the adversary acquires a login tuple of U in S:

$$(H(U), n, H(0, n, P, S), H^2(1, n, P, S))$$

Moreover, an adversary cannot acquire such a login tuple unless the adversary succeeds in breaking into the secure database(s) where site S stores the login tuples of all its users – a hard task to perform!

Even if an adversary somehow succeeds in acquiring a login tuple of U in site S, this adversary cannot succeed in launching a user impersonation attack, a human mistake attack, or a Phishing attack against TLP between the browser of U and S.

1. **Security against User Impersonation:**
   If an adversary acquires a login tuple of user U in site S, and if this adversary uses this tuple to impersonate a browser of user U and execute TLP in order to log into site S, then the adversary's attempt to log into site S will fail.

2. **Security against Human Mistakes and Phishing Attacks:**
   If an adversary acquires a login tuple of user U in site S, and if this adversary uses this tuple to establish an adversarial web site S' whose domain is different from that of S, and if user U requests that his browser executes TLP and logs into site S', then the login attempt will fail.

Still, if it is possible for an adversary to acquire a login tuple of user U in site S, then this adversary can launch successful Pharming and eavesdropping attacks against TLP between the browser of U and S. To protect against this possibility, the login tuple of U in S is partitioned into two subtuples:

$$(H(U), n, H(0, n, P, S)) \text{ and } (H(U), H^2(1, n, P, S))$$

Each subtuple is stored in a different database. Thus, the first subtuple is stored in database 0 and the second subtuple is stored in database 1. When user U initially registers in site S, the browser of U generates the first login tuple and sends it to site S. Site S divides the received login tuple into two subtuples and stores the first subtuple in database 0 and stores the second subtuple in database 1.

Later when user U attempts to log into site S, site S forwards each of the messages that S receives from U's browser to the appropriate database so that this database can process the message and return a reply to S which forwards the reply back to U's browser. For example, when S receives the Hello(U) message from U's browser, S selects a nonce SN at random and sends both U and SN to database 0 which prepares a Hello-Reply(n, $[H(0, n, P, S)] < SN >$) message and returns it to S which forwards it back to U's browser. Thus no one, not even site S, gets to keep track of the subtuples stored in the two databases 0 and 1.

Because the two subtuples of user U in site S are continuously changing, it is reasonable then to assume that an adversary, who attempts to acquire the two subtuples of the same user U from the two databases 0 and 1, will do so at different times and will end up with two unsynchronized subtuples of the following form:

$$(H(U), n, H(0, n, P, S)) \text{ and } (H(U), H^2(1, n', P, S))$$

Fortunately, if an adversary who succeeds only in acquiring two unsynchronized subtuples of user U in site S, then this adversary cannot succeed in launching a Pharming or eavesdropping attack against TLP between the browser of U and S.

3. **Security against Pharming Attacks:**

   If an adversary acquires two unsynchronized subtuples of user U in site S, and if this adversary uses these subtuples to establish an adversarial site S' whose domain is the same as that of site S, and if the DNS of U's browser is manipulated so that U's browser is directed to site S' instead of S, and if U requests that his browser executes TLP and logs into site S, then the login attempt will fail.

4. **Security against Eavesdropping:**

   If an adversary acquires two unsynchronized subtuples of user U in site S, and if this adversary attempts to eavesdrop on the TLP communication between

U's browser and S and obtain the cookie that is sent (encrypted) in the Login-Reply message of TLP, then the eavesdrop attempt will fail.

It is straightforward to prove that TLP satisfies the above four properties 1 through 4. To prove that TLP satisfies each of these properties, it is sufficient to identify a data item that the adversary will need, but will not be able to compute from its acquired data, in order to complete executing TLP successfully.

To prove that TLP satisfies Property 1, note that the adversary will not be able to compute the data item $H(1, n, P, S)$, even though it has acquired a login tuple of user U in site S, that is needed to compute the Login Message of TLP.

To prove that TLP satisfies Property 2, note that the adversary will not be able to compute the data item $H(0, n, P, S')$, even though it has acquired a login tuple of user U in site S, that is needed to compute the Hello-Reply Message of TLP.

To prove that TLP satisfies Property 3, note that the adversary will not be able to compute the data item $H^2(1, n, P, S)$, even though it has acquired two unsynchronized subtuples of user U in site S, that is needed to decrypt the Login Message of TLP and obtain BN.

To prove that TLP satisfies Property 4, note that the adversary will not be able to compute the data item BN, even though it has acquired two unsynchronized subtuples of user U in site S, that is needed to decrypt the cookie in the Login-Reply Message of TLP.

3.6 User Interface of TLP

As a proof of concept, we have developed a prototype of our Transport Login Protocol TLP. The browser side of our prototype is developed on the Firefox browser using the two technologies of Javascript and HTML. The web site side of our prototype is

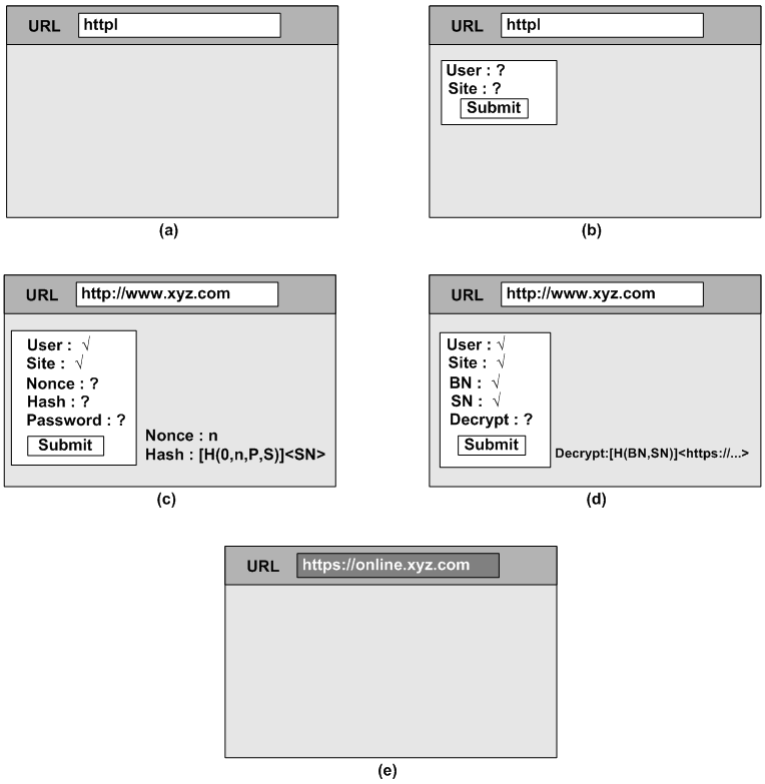


Figure 3.2: User Interface of TLP

developed on the Tomcat web server using four technologies: Java, HTML, the JSP (Java Server Page) technology, and the MySQL database technology. Note that the MySQL database technology is used to manage the login tuples, of all users, that are stored in the web site.

We employed standard cryptography in our prototype. In particular, we employed the Secure Hash Algorithm SHA-1 for secure hash, and employed the Advanced Encryption Standard AES for symmetric key cryptography.

The guiding principle in our prototype is to ensure that the user never enters his TLP universal password into a web page that is supplied by a web site, but he can enter his password into a local web page that is supplied by his own browser. It turns out that this principle is hard to fulfill in our prototype in the light of the

"Same Origin Policy" that is adopted by the Javascript technology. At the end, however, we were able to fulfill this principle by designing a novel user interface for our prototype. We discuss this user interface next.

Figure 3.2 details the four steps that need to be taken by a user to log into a web site `www.xyz.com`.

1. The user first enters `httpl` into the URL box of his browser and pushes $<$ *return* $>$; see Figure 4a. This causes a display of the local page `httpl` to appear as a small window on the left site of the screen; see Figure 4b.

2. The user enters his username and the name of the site `www.xyz.com` into page `httpl` then clicks on the $<$ *submit* $>$ button in this page. This causes page `httpl` to execute, update its own display, and send a Hello message (the first message in TLP) to site `www.xyz.com` which replies by sending back the web page `http://www.xyz.com`. This page contains the two fields, named nonce and hash, of the Hello-Reply message (the second message in TLP); see Figure 4c.

3. The user copies the values of the two fields nonce and hash from the displayed page `http://www.xyz.com` and enters them into page `httpl`. The user then enters his password into page `httpl` and clicks on the $<$ *submit* $>$ button of this page. This causes page `httpl` to execute, update its own display, and send a Login message (the third message in TLP) to site `www.xyz.com` which replies by sending back a new web page `http://www.xyz.com`. This new page contains one field, named decrypt, of the Login-Reply message (the last message in TLP); see Figure 4d.

4. The user copies the value of field decrypt from the displayed page `http://www.xyz.com` and enters it into page `httpl`. The user then clicks on the $<$ *submit* $>$ button of page `httpl`. This causes page `httpl` to execute, compute the next

HTTPS page, say page `https://online.xyz.com`, that needs to be called into the browser, and redirects the browser to call this page using TLS and display it on the screen. Note that in this case the browser assigns the displayed HTTPS page a brown classification, and so the background color of the URL box of the displayed page becomes brown as shown in Figure 3.2.

Because the browser has classified the displayed page `https://online.xyz.com` brown, then if the user clicks on any link (of an HTTPS page) in page `https://online.xyz.com`, then the browser will classify the newly called page brown as well.

3.7 Related Work

Despite advances in security technologies, the Internet is plagued by vulnerabilities. These vulnerabilities exploit technical weaknesses as well as the propensity of humans to lowering their guards in routine transactions. In our daily web surfing experiences, homograph attacks [42] work very well for this reason. For the same reason, security indicators in browsers are not enough to prevent a user from accessing web pages hosted by malicious servers and alternative approaches are required [26]. While security mechanisms have been relatively successful in protecting systems from attacks that exploit technical loopholes or syntactic attacks [107], dealing with semantic attacks has proven much more difficult. As defined by Bruce Schneier [107], semantics attacks are attacks that depend on the way humans assign meaning to content. The difficulty of dealing with semantic attacks is due to the fact that most security systems depend on perfect human behavior that is always vigilant. In practice, security is not a user's primary concern and security checks and warnings can be considered by users as getting in the way of their use of applications. Thus, user errors result in security failures [116]. The total number of unique Phishing reports submitted to Anti-Phishing Working Group (APWG) in

November 2007 was 28,074 [48]. Gartner says that Phishing attacks are estimated to have cost $2.8 Billions in the year 2006 alone [78].

3.7.1 Client-based anti-Phishing tools

In order to prevent Phishing in the client side, many anti-Phishing tools are developed. These anti-Phishing tools are based on blacklists, whitelists, and heuristics. Netcraft [89] mainly depends on blacklists and it cannot detect a new Phishing site. Spoof Guard [23] is based on heuristics and it uses domain name, url, link, and image to evaluate the likelihood that a given page is part of a Phishing attack. SpoofGuard has high catch rate of 90% but also has high false positive rate of 42% [127]. CANTINA [128] is called a content-based approach since it is not only based on heuristics similar to Spoof Guard but also based on TF-IDF (Term Frequency and Inverse Document Frequency) information. It succeeds to reduce the false positives compared to Spoof Guard. Security toolbars such as SpoofStick [110], Netcraft Toolbar, Paypal Trustbar [53], eBay Account Guard [29], and SpoofGuard are designed for humans to use, but usability studies found them all ineffective to prevent Phishing attacks [117].

3.7.2 Password-based approach

Password Hash [102] is a browser extension that allows users to log into multiple sites transparently with a universal password. The browser extension applies a cryptographic hash function to a combination of the plaintext password, domain name, and a salt. Thus, the break of one system does not lead to that of other sites. Usability study of Password Hash [17] involving 26 users showed that password hash suffers from major usability problems. In addition to that, if password hashing is based on the domain name, it is vulnerable to Pharming attacks [92, 111].

3.7.3 Server-based anti-Phishing

OpenDNS [93] is a way to filter Phishing sites at DNS levels. Thus, it is a black-list based approach in DNS servers. Phishtank [94] is the online Phishing database which feeds this information to OpenDNS servers. PhishBouncer [79] uses HTTPS proxying and attribute-based checks to defend against Phishing attacks. Phish-Bouncer is based on whitelists, blacklists and heuristics. But it is still vulnerable to Pharming attacks and requires a HTTPS proxy to be deployed in the Internet.

3.7.4 Cookie-based anti-Phishing

Though cookie-based approaches are effective in preventing Phishing attacks, they have limitations since cookies can be easily purged and cannot be a permanent solution for Phishing attacks. Temporary Internet Files (TIFs) are called cache cookies and are used as authenticators to protect from Phishing and Pharming attacks [63]. Active Cookies [64] rely on a new protocol that tags cookies with a specific, valid IP address and the channel redirected by a server and active cookies are fetched by the server to verify the client. But, it is vulnerable to IP based attacks. Adversaries can corrupt DNS [8] and BGP [90]. In order to resolve the vulnerabilities of the most vital function, DNS in the Internet, DNS Security Extensions (DNSSEC) are proposed [4–6]. However, DNSSEC has obstacles to deploy in the Internet due to compatibility, scalability, and the ownership of a root key. Locked Cookies [65] are cookies that are bound to the originating server's public key. Clients verify the server by comparing the public key in the locked cookie and the public key presented by the server. Web Server Key Enabled Cookie (WSKE) [77] bind cookies to domain names and public keys and similar to locked cookies.

3.7.5 Interface-based anti-Phishing

Interface-based anti-Phishing is the most popular approach to protect from Phishing attacks. PassMark [108] is a way to share a secret between a user and a web site and helps the user authenticate the web site. The user provides the web site with a shared secret such as an image and/or a passphrase in addition to his regular password. Some examples include site key [10] of Bank of America and Yahoo Sign-in Seal [121]. Site key consists of a unique image that users chose, an image title, and three challenge questions. By showing the unique image to users, users verify the server. However, site key is vulnerable to man-in-the-middle attack [123]. Moreover, the efficacy of security indicators such as HTTPS indicators and site-authentication images are ineffective where users ignore HTTPS indicators and site-authentication images [106]. Yahoo Sign-in Seal is a secret message or image that users create to protect Yahoo accounts. It is similar to site key but it provides a way to create a customized text message with colors or images that users chose. Challenge questions are not completely free from security problems since common knowledge such as mother's maiden name can be collected in the Internet [47].

Dynamic Security Skins (DSS) [25] uses the trusted password window with a background image and SRP to authenticate a user and a server. Hash Visualization [24] is used to ensure that users are visiting authenticated web pages in DSS. Web Wallet [118] is a browser sidebar designed for users to submit their sensitive information. Users are required to press a dedicated security key on the keyboard to activate Web Wallet. However, Web Wallet itself can also be phished and users might not use Web Wallet to submit sensitive data.

3.7.6 Certificate-based authentication

Certificates seem to be a panacea for security. However, that PKI can provide security is a popular falsehood [30]. PKI is more complex than what most people

think. The problem arises when PKI is not well implemented and practiced. Certificate Authorities like VeriSign might issue erroneous certificates. VeriSign issued two VeriSign Class 3 code-signing digital certificates to an individual who fraudulently claimed to be a Microsoft employee [80]. Though VeriSign revoked those certificates, those certificates might be used if certificate validation is not implemented. The certification path validation algorithm is the algorithm which verifies that a given certificate path is valid under a given public key infrastructure [55]. This is complex, assuming there are a number of intermediaries between the subject and the root. Delegated Path Validation (DPV) [95] and Server-based Certificate Validation Protocol (SCVP) [39] facilitates the validation of certificates.

OCSP (Online Certificate Status Protocol) [87] is an Internet protocol to obtain the revocation status of an X.509 digital certificate. Thus, clients must implement OCSP to check the status of certificates, but it was rarely deployed in the past.

To remedy Phishing problems, Extended Validation Certificate [36] was proposed. In order to highlight secure sites in IE7, it proposed to use different colors in URL address box. Unfortunately, this new technique is still vulnerable to picture-in-picture Phishing attacks and not promising to rectify Phishing problems [61].

3.7.7 Strong Password Protocols

TLS [14] provides privacy communication between two parties, but it does not ensure authentication. Strong password protocols provide authentication and prevents eavesdropping and impersonation. Encrypted Key Exchange (EKE) [12], Augmented EKE (AEKE) [13], Simple Password Exponential Key Exchange (SPEKE) [58], Password Derived Moduli (PDM) [66], and Secure Remote Password (SRP) [119,120] are Strong password protocols. For mutual authentication, TLS-SRP [113] and TLS-PSK (Transport Layer Security Pre-shared Key) [32] must be used. Some of the

objectives of TLP are achieved by the SRP, but not all the objectives of TLP are achieved by SRP. For example, TLP supports universal passwords but SRP does not. Also, TLP is simpler than SRP. For example, TLP is based on standard functions for performing secure hash and symmetric encryption and decryption, whereas SRP is based on a function for performing exponentiation.

3.8 Concluding Remarks

In this Chapter, we present a comprehensive proposal to counter human mistakes and Phishing and Pharming attacks that may occur when a user attempts to log in a secure web site. Our proposal is based on two ideas. First, we introduce a new classification, brown, of secure HTTPS web pages. When the browser of a user U classifies a displayed page brown, user U should conclude that the displayed page is secure and can enter his sensitive data into it. Second, we design a new login protocol, named TLP, that is secure against human mistakes and Phishing and Pharming attacks. The browser of a user U uses TLP to classify a displayed page brown according to two rules:

1. The displayed page is called into the browser using TLP.

2. The displayed page is called into the browser, using TLS, from within another brown page that was displayed earlier on the browser.

Note that TLP is not intended to replace TLS. On the contrary, our vision assigns complementary roles to be played by TLP and TLS: TLP can be used first to securely log into a web domain, then TLS can be used later to securely go from one web site to another within the logged-in domain.

Note also that some mildly secure web domains may feel that they are in no danger of facing Phishing or Pharming attacks because adversaries have little incentive to launch such attacks against these domains. (Examples of such domains

are those that host electronic reviewing and handling of submitted papers to conferences and journals.) These web domains can keep on employing TLS, as they do presently, both for logging in a domain and for going from one web site to another within this domain.

A nice feature of TLP is that a user can use the same username and same (TLP universal) password to securely log into any web site in the Internet. This means that the user need only to memorize one username and one password for all web sites. Therefore it is reasonable to demand that each user chooses a long string, say of sixteen characters, to be his TLP universal password. And so TLP becomes naturally secure against online and offline dictionary attacks.

Chapter 4

TPP: The Two-way Password Protocol

The need for secure communication in the Internet has led to the widespread deployment of secure application-level protocols. The current state-of-the-art is to use TLS, in conjunction with a password protocol. The password protocol, which we call a one-way password protocol (OPP), authenticates a user to a server, using a particular secret called the password. TLS has two functions: (1) It ensures secure communication between a client and a server and (2) It allows a user to authenticate a server. The first function effectively provides a secure channel for end-to-end communication between a client and a server. However, the second function is frequently compromised by a variety of Phishing attacks. In this Chapter, we address this problem by developing a password protocol which we name the Two-way Password Protocol (TPP). TPP, when used in conjunction with TLS, ensures that users correctly authenticate servers, and are protected from Phishing attacks. The first contribution of this Chapter is to develop a protocol, called the Universal Password Protocol (UPP), which ensures that a user's password is kept safe even in the case of a successful Phishing attack. However, it may be noted that a user, after logging

in, frequently shares other secrets (such as credit card numbers) over the secure connection, and UPP cannot protect these. Our second contribution is to build on UPP and develop, first, the Two-Way Password Protocol (TPP), and finally an improved version named the Dynamic Two-Way Password Protocol (DTPP), which ensures that both a server and a client are properly authenticated to each other. This ensures the security of all secrets which should be known only to the client and the server, including, of course, the password. These protocols were introduced and published in [20].

A problem of immense importance in any scenario where users with different privileges use a system is authentication. Clearly, it is necessary for a party that controls access to a resource to verify the identity of the party requesting access; otherwise, there is no way to determine which requests to allow, and which to deny. Thus, authentication is essential whenever an entity provides other specific parties with access to special resources – for example, by sharing secrets with them. A major application of this is seen in the use of secure remote protocols in the Internet, where the resources provided range from email to auctions to banking. In general, authentication on the Internet involves two parties: a user and a server.

The current state-of-the-art in providing online authentication involves a combination of two protocols: the Transport Layer Security protocol, or TLS [27], and a simple password protocol, which we have named the One-Way Password Protocol OPP. TLS certifies to the user that the server is not *spoofing*, i.e. presenting a false address; it also ensures that a user and a server have a cryptographically secure channel of communication. OPP authenticates the user to the server.

Unfortunately, this combination is not enough to ensure security against some classes of attack. For example, by simple human error, or by use of a Phishing attack, a user U may end up going to a malicious server M (say eebay.com) instead of the actual server S she has a relationship with (in this case, ebay.com). TLS does

not protest – the site is not spoofing; the address bar indeed says eebay.com, which is correct. OPP is useless here; it authenticates a user to a server, not a server to a user. Thus, the adversary M can harvest secrets from U – most importantly the password authenticating U to S, but also other secrets she might share with S, such as credit card numbers, home addresses, and so on.

Earlier work in the area of providing security against such attacks, such as the TLP protocol, have focused on how to strengthen TLS so that it can defend against these attacks. In the system developed by Choi et al. [22], TLP must be used for the first login into a secure page; TLS can be used to authenticate pages subsequently reached from a secure page.

In this Chapter, we study the problem of ensuring secure authentication that is robust against Phishing attacks (and related problems such as user error). We achieve this goal using the standard TLS protocol, simply by modifying the password protocol so it achieves two-way authentication between a server and a user. Our solution is developed step by step. We start with OPP and modify it to the server password protocol (passwords for a user on a server and for a server on a user). This simple protocol can be defeated by using a Phishing attack in conjunction with a Man-In-The-Middle attack. Hence, we modify it and develop the Universal Password Protocol UPP, which is adequate for the purpose of protecting passwords. (It may be noted in passing that UPP also solves the problem of password reuse – the user only has to remember one password for access to all secure sites, without the problem of reusing a password at multiple servers.) However, we note that even UPP can be beaten; a malicious server can simply log the user in, and steal other secrets from her, when it fails to obtain her password. In answer to this problem, we develop the Two-Way Password Protocol TPP and improve it to our final version, the Dynamic Two-Way Password Protocol DTPP. DTPP ensures that all the secrets shared between a user and a server – and in particular the user's password on the

server – are secure from Phishing attacks.

We begin with a detailed description of TLS and the current state of the art, in the following Section.

4.1 Background

Before a user U can communicate with a secure website S over the web, both U and S need to authenticate one another by executing two protocols: the standard TLS protocol [27] which allows U to authenticate S, and a usual one-way password protocol which allows S to authenticate U. For completeness, in this Section we briefly review the standard TLS protocol.

A certificate of a website S is a data structure that has the following format:

$$(R, S, K_S, t, sig)$$

where R is an issuer, S is a website, K_S is a public key of S, t is an expiration date, sig is a signature.

This certificate can be viewed as the statement "R asserts that the public key of website S is K_S, from now until date t." When a user U receives this certificate, U needs to perform two checks. First, U needs to check that the certificate is current by checking that the expiration date t has not yet been reached. Second, U needs to check that the certificate is valid by using the signature of the certificate, as discussed below, to validate that R has indeed issued the certificate. If C concludes that the certificate is both current and valid, then C accepts that K_S is the public key of S.

The signature of the certificate is computed by the issuer R as follows:

$$sig := K_R^{-1} < H(R, S, K_S, t) >$$

where K_R^{-1} is the private key of R, H is a standard secure hash function, $H(R, S, K_S, t)$

is the result of applying the hash function H to the concatenation of the four items R, S, K_S, and t in the certificate, and $K_R^{-1} < H(R, S, K_S, t) >$ is the encryption of $H(R, S, K_S, t)$ using the private key K_R^{-1} of issuer R.

When a client C receives this certificate, C needs to use the signature of the certificate to validate that R issued the certificate. To perform this check, client C needs to know a priori the public key K_R of issuer R. (Public keys from trusted Certification Authorities, for example Verisign, come pre-loaded with all browsers in use today.) User C performs this check as follows:

1. Client C decrypts the signature of the received certificate using the public key K_R of issuer R.

2. Client C applies the secure hash function H to (the concatenation of) the four fields R, S, K_S, and t in the certificate.

3. If the values computed in the two previous steps are equal, then user C concludes that R has indeed issued the received certificate. Otherwise, client C concludes that R has not issued the certificate.

We will now present the execution of a simple scenario of the TLS protocol. A client C and a server S are executing the protocol, using the certificate of S.

$$
\begin{aligned}
C \rightarrow S: \quad & \textit{client-hello}(nc) \\
C \leftarrow S: \quad & \textit{server-hello}(ns), \\
& certificate((R, S, K_S, t, sig)) \\
C \rightarrow S: \quad & \textit{key-exchange}(K_S < pms >), \\
& finished(H(nc, ns, pms, ms)) \\
C \leftarrow S: \quad & finished(H(ns, nc, pms, ms))
\end{aligned}
$$

where nc is a nonce selected at random by client C and sent in the clear to website S, ns is a nonce selected at random by website S and sent in the clear to client

C, pms is a premaster secret selected at random by client C and sent in private to website S after it is encrypted by the public key K_S of S, and ms is the master secret computed by both client C and website S using the three values nc, ns, and pms.

The $key\text{-}exchange(K_S < pms >)$ message, sent from C to S in the third step of this scenario, is intended to challenge S, if it is indeed S, to use its private key K_S^{-1} to decrypt the message, obtain the premaster secret pms, and use pms to compute the master secret ms. When client C checks, in the fourth step, that S was able to correctly compute ms, C knows that it is indeed communicating with S.

The $finished(H(nc, ns, pms, ms))$ message, sent from C to S in the third step of this scenario, is intended to assure website S that client C was able to compute the master secret ms correctly. Similarly, the $finished(H(ns, nc, pms, ms))$ message, sent from S to C in the fourth step of this scenario, is intended to assure client C that website S was able to compute ms correctly.

At the end of this execution, the following two outcomes are achieved.

1. Client C knows that it is indeed communicating with website S.

2. Both C and S agree on a master secret ms that they can use to encrypt and decrypt all the messages that they need to exchange next.

Note that the authentication is not symmetric; server S does not know the client with whom it is communicating. In order to make up for this shortcoming, TLS is usually paired with a simple password protocol to authenticate client C to server S.

Unfortunately, this arrangement is not adequate to provide security. In the next Section, we show that despite the security provided by the TLS protocol, there exist attacks that can circumvent the security provided by a combination of TLS

with normal password authentication.

4.2 The One-Way Password Protocol

In this Section, we argue that the authentication procedure that is based on the standard TLS protocol and the traditional one-way password protocol are vulnerable to Phishing attacks.

After executing the TLS protocol as in Section 4.1, and in order for S to know user U with whom it is communicating, U and S execute the following two steps of the one-way password protocol. (Note that the messages exchanged in these two steps are encrypted using the master secret ms that is computed in the TLS protocol.)

$$U \leftarrow S: \quad ms < \text{enter (user id, password)} >$$
$$U \rightarrow S: \quad ms < (U, pw) >$$

Prior to executing these two steps, user U has registered the pair $(U, H(pw))$ in website S where pw is a password of U. Thus, when S receives the message $ms < (U, pw) >$ from U, S concludes that it is indeed communicating with user U.

Clearly, OPP ensures that the user is authenticated to the server. TLS ensures that the server is authenticated to the user's browser; when the URL for site S is displayed in the location bar of the browser, the user is indeed at site S.

However, the combination of OPP and TLS has one subtle weakness. Authenticating the server to the user's browser is not the same as authenticating the server to the user. In fact, an adversary M can defeat this security measure simply by not spoofing (i.e. M does not claim to be at the URL of server S) and using other means to make the user U associate M with S.

We will now describe an example of a Phishing attack that can defeat the two authentication protocols, the TLS protocol and the one-way password protocol,

discussed above.

Simple Phishing Attack:

A user U receives the following email:

> "For being a good customer of the website `https://www.ebay.com`, we offer you a special deal. Please log into the website `https://www.specialdeals.com` to check out our great deal to you."

From now on, we refer to the website `https://www.ebay.com` as website S, and we refer to the website `https://www.specialdeals.com` as website M.

Excited by this email, user U proceeds to log into website M. First, the TLS protocol is executed between client C of user U and website M so that U can be certain that it is indeed communicating with M. Second, M sends to U the message $ms <$ enter (user id, password) $>$, but this message is displayed on a webpage that has the same logo and graphics as that of website S. Third, user U enters into the displayed webpage his user id U and his password pw, which U has registered earlier in website S. Fourth, user U receives from website M a webpage that offers U to purchase a good collection of DVDs for a cheap price and instructs U to enter his credit card number if he is interested in purchasing this collection. Fifth, user U decides to purchase the DVD collection and enters his credit card number into the webpage.

Unfortunately for user U and website S, website M is not related in any way to website S. (The fact is that sites S and M belong to different domains, as site S belongs to the domain `ebay.com` and site M belongs to the domain `specialdeals.com`, should have implied that these two sites are not related.) In fact, M is an adversarial website that has just launched a successful Phishing attack against user U and website S and obtained the pair (U, pw), which user U has registered earlier in website M, along with the credit card number of user U.

After obtaining this information, website M can launch two more attacks against user U and website S.

1. A User Impersonation Attack:
   Using the pair (U, pw), M can successfully log into website S pretending to be user U.

2. An Identity Theft Attack:
   Using the credit card number of U, M can purchase many items over the web.

This Phishing attack is successful because neither the TLS protocol nor the one-way password protocol attempted to authenticate the fact that websites S and M are related to one another.

4.3 The Server Password Protocol

In the previous Section, we see clearly that the standard practice (of using OPP and TLS) can be broken by Phishing attacks. However, we note that the use of OPP in conjunction with TLS does in fact ensure that the following two conditions hold:

1. The server is in fact the server whose address is currently displayed in the user's address bar.

2. The user is authenticated to the server.

The problem is that the server is not properly authenticated to the user; user U can be sure that it is indeed communicating with server M, but has no way of knowing whether M is in fact associated with server S with which U has a relationship of trust.

We note that the user is properly authenticated to the server. This asymmetry is caused by the fact that OPP checks to make sure that the user has the correct password to log on to the server, but there is no corresponding check for the server.

Based on the above observation, it is natural to ask whether simply making the password protocol more symmetric would solve the problem. Just as the user is authenticated to the server by knowledge of a password, the server is authenticated to the user by knowledge of a secret called the *server password*. (For example, a server password can be a unique image, a phrase etc.) The user stores the server password on server S. In subsequent interaction, S authenticates itself to U by sending U its server password.

We name this protocol the server password protocol, and show its working below.

User U stores in website S the triplet:

$$(U, ps, H(pw))$$

where U is a user id, ps is a server password, and pw is a password of user U.

$U \leftrightarrow S$: execute TLS and compute ms

$U \leftarrow S$: $ms <$ enter user id $>$

$U \rightarrow S$: $ms < U >$

$U \leftarrow S$: $ms < ps,$ enter password $>$

$U \rightarrow S$: $ms < pw >$

Unfortunately, the attractive hypothesis that this protocol is robust against Phishing attacks is incorrect. We demonstrate that a Phishing attack, combined with a Man-In-The-Middle attack, succeeds in compromising the server password protocol.

Phisherman in the Middle Attack:

1.	$U \leftrightarrow$	M	: execute TLS and compute ms
2.		$M \leftrightarrow S$	: execute TLS and compute ms'
3.		$M \leftarrow S$	: $ms' <$ enter user id $>$
4.	$U \leftarrow$	M	: $ms \ <$ enter user id $>$
5.	$U \rightarrow$	M	: $ms < U >$
6.		$M \rightarrow S$	: $ms' < U >$
7.		$M \leftarrow S$	: $ms' < ps,$ enter password $>$
8.	$U \leftarrow$	M	: $ms < ps,$ enter password $>$
9.	$U \rightarrow$	M	: $ms < pw >$
10.		$M \rightarrow S$	: abort login procedure
11.	$U \leftarrow$	M	: $ms <$ enter credit card number $>$
12.	$U \rightarrow$	M	: $ms < cc >$
13.		M	: gets both password pw and credit card number cc

The reason for the insecurity of this protocol, is that both authentications (a user to a server and a server to a user) do not happen simultaneously. At some step, one party, the user or the server, has to take a "leap of faith" and go first, sending its secret (password or server password, respectively) to the other party before it is authenticated. In this case, it is the server that sends its server password to the user before it has seen the user password; consequently, the adversary M can obtain the server password from S and break the protocol, as shown above.

4.4 The Universal Password Protocol

In this section, we present a protocol, called the Universal Password Protocol, which ensures that the password of U on S cannot be stolen by Phishing attacks. The primary idea is that the user only has to remember one universal password; the password for a server is generated when needed.

User U stores in website S the triplet:

$$(U, ps, H(pw))$$

where U is a user id, ps is a server password, and the password pw is computed as follows.

$$pw := H(upw, ds)$$

where H is a standard secure hash function, *upw* is the *universal password* of user U, ds is the domain name of web site S (for example if S is the website `https://www.amazon.com`, then ds is `amazon.com`), and $H(upw, ds)$ is the application of function H to the concatenation of the universal password upw of user U, and the domain name ds of server S.

The execution of the universal password protocol proceeds as follows:

$U \leftrightarrow S$: execute TLS and compute ms

$U \leftarrow S$: $ms <$ enter user id $>$

$U \rightarrow S$: $ms < U >$

$U \leftarrow S$: $ms < ps,$ enter password $>$

$U \rightarrow S$: $ms < pw >$

It may be noted that the user sends the server its password to authenticate itself, but the server stores only the secure hash of the password. The reason for this measure is to ensure that, even if the server is compromised – for example, by disgruntled employees – and the store of hashed passwords is stolen, the attacker cannot start using this database of stolen passwords to impersonate U.

We can now make a very interesting observation. As TLS prevents spoofing, U knows the domain name of M, and will use dm rather than ds to compute the password sent to server M; consequently, M cannot learn the password of U on S

by means of Phishing attacks.

However, this protocol, while perfectly adequate for the purpose of protecting the password of U, does not serve to protect any other secrets shared between U and S.

To see why, we consider the following subtle Phishing attack.

Persevering Phisherman Attack:

1.	$U \leftrightarrow$	M	: execute TLS and compute ms
2.		$M \leftrightarrow S$	: execute TLS and compute ms'
3.		$M \leftarrow S$	: $ms' <$ enter user id $>$
4.	$U \leftarrow$	M	: $ms <$ enter user id $>$
5.	$U \rightarrow$	M	: $ms < U >$
6.		$M \rightarrow S$	: $ms' < U >$
7.		$M \leftarrow S$	: $ms' < ps,$ enter password $>$
8.	$U \leftarrow$	M	: $ms < ps,$ enter password $>$
9.	$U \rightarrow$	M	: $ms < H(upw, dm) >$
10.		$M \rightarrow S$	: abort login procedure
11.	$U \leftarrow$	M	: $ms <$ enter credit card number $>$
12.	$U \rightarrow$	M	: $ms < cc >$
13.		M	: gets credit card number cc

On close observation, we see that the weakness in UPP is again due to the problem that authentication of U to S and of S to U does not happen in a single step, and consequently, there remains a possibility that the server password of S can be stolen.

However, we have seen in UPP that, by incorporating the domain name ds of server S into the password, we can protect the password from being stolen by the adversary; M can get a password, but it will not get the password of U on S. It is

natural to ask if the same idea, using ds in the server password of S, can protect the server password from being stolen and used by M. We develop this idea in the following section.

4.5 The Two-Way Password Protocol

In the previous Section, we demonstrated that while UPP is adequate for protecting passwords, the security of a system using UPP can still be compromised by an adversary using a more subtle attack, which we call Persevering Phisherman attack. A secure session involves not only the password, but also other secrets; even if the adversary M cannot acquire the password, it can acquire the other secrets shared between U and S – such as the credit card number of U.

In this Section, we address this vulnerability to develop a more advanced version of our protocol, which we call the Two-Way Password Protocol (TPP). This protocol ensures that, unlike in UPP, the malicious server M cannot simply pass on to U a server password from S. Thus, TPP (in conjunction with TLS to prevent spoofing) protects not only the password, but also any other secrets shared between U and S.

The fundamental insight behind this protocol is the fact that the hash of the user's password $H(pw)$, stored on the server, can in fact itself be used as a server password. In our exposition on UPP, we showed how password pw can be made site-specific by incorporating the server domain name ds. By using $H(pw)$ as the server password of S, we make the server password domain-specific also. To authenticate itself to U, the adversary server M needs to produce the corresponding server password $H^2(upw, dm)$. But M cannot obtain this server password; its value is not present on server S (because S stores $H^2(upw, ds)$, not $H^2(upw, dm)$) and M cannot calculate it without knowledge of upw (which U does not share). Hence we solve the problem of the server M stealing the server password of S and authenticating

itself to the user.

User U stores in website S the pair:

$$(U, H(pw))$$

where U is a user id, and pw is computed as discussed in UPP in the previous Section.

$$pw := H(upw, ds)$$

$$\begin{aligned}
U \leftrightarrow S &: \text{execute TLS and compute } ms \\
U \leftarrow S &: ms < \text{enter user id} > \\
U \rightarrow S &: ms < U > \\
U \leftarrow S &: ms < H^2(upw, ds), \text{enter password} > \\
U \rightarrow S &: ms < H(upw, ds) >
\end{aligned}$$

The Failure of Phishing Attacks:

1.	$U \leftrightarrow$	M	: execute TLS and compute ms
2.		$M \leftrightarrow S$	: execute TLS and compute ms'
3.		$M \leftarrow S$	: $ms' <$ enter user id $>$
4.	$U \leftarrow$	M	: $ms <$ enter user id $>$
5.	$U \rightarrow$	M	: $ms < U >$
6.		$M \rightarrow S$	: $ms' < U >$
7.		$M \leftarrow S$	: $ms' < H^2(upw, ds)$, enter password $>$
8.	$U \leftarrow$	M	: The attack fails at this point since M cannot compute $ms < H^2(upw, dm) >$ to send it to U

4.6 The Dynamic Two-Way Password Protocol

In the previous Section, we demonstrated TPP, which is secure against even sophisticated Phishing attacks. For all practical purposes, TPP achieves our goal of being immune to Phishing attacks.

However, unlike some advanced login protocols such as TLP [22], TPP does not have the feature of one-time login data. In TLP, the login data needed to authenticate U to S is updated on each login, so even if an adversary manages to acquire the login data of U to S, the stolen data becomes useless after the next login of U into S.

Under the assumption of secure TLS, the password is secure in TPP – it is only sent to the authenticated server S, and is encrypted to make sure that it cannot be stolen by an eavesdropping attack. However, in practice, there do exist attacks that break the security assumptions of TLS; for example, the root certificate

authorities accepted by browsers are not always trustworthy. A password protocol cannot protect against such an attack (where TLS is broken and the attacker M can spoof as S). However, in order to minimize the damage if even such an attack is carried out, we incorporate into TPP the additional feature of one-time login data. The secrets needed to authenticate U to S, and S to U, are updated on each login; hence, even if the adversary does manage to steal the password of U, the stolen password is only useful until the next time U logs into S. We call this final version of our protocol the Dynamic Two-Way Password Protocol (DTPP).

User U stores in website S the triplet:

$$(U, n_i, H(pw_i))$$

where U is a user id, n_i is (the i'th value of) a nonce chosen by the user, and pw_i is

$$H(upw, n_i, ds)$$

where upw is the universal password of user U and ds is the domain name of server S.

$$\begin{aligned}
U \leftrightarrow S &: \text{execute TLS and compute } ms \\
U \leftarrow S &: ms < \text{enter user id} > \\
U \rightarrow S &: ms < U > \\
U \leftarrow S &: ms < n_i, H^2(upw, n_i, ds), \text{enter password} > \\
U \rightarrow S &: ms < H(upw, n_i, ds), n_{i+1}, H^2(upw, n_{i+1}, ds) >
\end{aligned}$$

We see that the working of the Dynamic Two-way password protocol (DTPP) is almost exactly similar to that of the Two-way password protocol (TPP). The main difference is that the password is some fixed pw, but pw_i; it varies with each login.

Server S stores the last value of the nonce n_i and the corresponding $H(pw_i)$. When user U tries to log into, she is given her n_i as well as the corresponding

$H(pw_i)$. As U knows H, upw, n_i, and ds, she can check that $H(pw_i)$ is correct, and authenticate the server. Now she chooses the next value of the nonce to be n_{i+1}. In the last step, she sends to the server the password $pw_i = H(upw, n_i, ds)$ (so user is authenticated to server), and the pair n_{i+1} and $H^2(upw, n_{i+1}, ds)$, i.e. $H(pw_{i+1})$. S replaces the stored n_i and $H(pw_i)$ with n_{i+1} and $H(pw_{i+1})$; these values will be used the next time a user tries to log in with user name U. Thus, the password and server password change with every use in this protocol.

4.7 Related Work

Secure remote authentication of parties over the Internet is an extremely important problem, and has been the focus of considerable research. In this section, we discuss a few relevant protocols, and specify the contribution of this Chapter in the context of earlier work.

As TPP is a password protocol, it is most natural to consider it in the context of earlier password protocols. From the development of the protocol, it is clear that the most interesting feature of TPP is its immunity to Phishing attacks, which break ordinary password protocols, simple challenge-handshake authentication protocols such as site key [10] and message digest protocols [37], and hash-based protocols [49].

However, TPP, thanks to its use of a universal password, has several other highly desirable features. Early password protocols such as Lamport's [75] and Rubin's [103] one-time password protocols are forced to use a list of passwords, which the client uses one time only, to guard against the threat of eavesdropping attacks. This, of course, leads to the serious inconvenience of having to remember and register a huge list of passwords. On the other hand, protocols that depend on one central server to authenticate clients for multiple servers [81] have a single point of failure and require a high cost of integration. TPP circumvents all of these problems; it ensures that login data is for one-time use only, but requires the user

to remember only one (strong) password, and, unlike the Passpet system [122], does not require any external server for authentication.

Another important feature of TPP is that it uses no exotic computation such as modular exponentiations etc; the only required computation, (in addition to the standard encryption/decryption done by TLS) is the computation of one secure hash at the client and one at the server. Thus, it does not use any non-standard operations or require much processing power, unlike other strong password protocols such as EKE [12] and SRP [120]. Moreover, as it is built to be used in conjunction with TLS, it benefits directly from improvements to TLS. For example, TLS is currently being upgraded to use SRP as an underlying layer. This will improve the security of TLS, and thus strengthen a system running TPP, as any such system also runs TLS.

The closest ancestor to TPP is our own earlier protocol SPP [45]. However, SPP is a single password protocol, whose aim is simply to safeguard the user's single (universal) password. Thus, SPP is vulnerable to subtle attacks that try to steal other secrets besides the user's password (such as the Persevering Phisherman attack). TPP provides complete protection of all secrets from Phishing attacks.

4.8 Concluding Remarks

Standard authentication over the web, using TLS and a password protocol, is easily compromised by user error and Phishing attacks. In this Chapter, we present a strong protocol, UPP, which ensures that the user's password cannot be compromised. Next, building on UPP, we develop TPP and finally DTPP, a password protocol which (in conjunction with TLS) provides mutual authentication between client and server, and protects all shared secrets between client and server from Phishing attacks. It may be noted that this protocol is fairly lightweight; it takes only four messages (the original password protocol itself takes two), and imposes

little additional computational or storage load on the client or on the server. We suggest that, given the widespread prevalence of Phishing attacks [114], there is good reason to deploy the TPP protocol and replace the one-way passwords that are used on the Internet today.

Chapter 5

BKP: The Best-Keying Protocol in Sensor Networks

Many sensor networks (especially networks of mobile sensors or networks that are deployed to monitor crisis situations) are deployed in an arbitrary and unplanned fashion. Thus, any sensor in such a network can end up being adjacent to any other sensor in the network. To secure the communications between every pair of adjacent sensors in such a network, each sensor x in the network needs to store $n-1$ symmetric keys that sensor x shares with all the other sensors, where n is an upper bound on the number of sensors in the network. This storage requirement of the keying protocol is rather severe, especially when n is large and the available storage in each sensor is modest. Earlier efforts to redesign this keying protocol and reduce the number of keys to be stored in each sensor have produced protocols that are vulnerable to impersonation, eavesdropping, and collusion attacks. In this Chapter, we present a fully secure keying protocol where each sensor needs to store $(n+1)/2$ keys, which is much less than the $n-1$ keys that need to be stored in each sensor in the original keying protocol. We also show that in any fully secure keying protocol, each sensor needs to store at least $(n-1)/2$ keys. Our keying protocol to store

$(n+1)/2$ is the best possible secure protocol and was published in [18].

Many wireless sensor networks are deployed in arbitrary and unplanned fashion. Examples of such networks are networks of mobile sensors [56] and networks that are deployed in a hurry to monitor evolving crisis situations [105] or continuously changing battlefields [57].

In any such network, any deployed sensor can end up being adjacent to any other deployed sensor. Thus, each pair of sensors, say sensors x and y, in the network need to share a symmetric key, denoted $K_{x,y}$, that can be used to secure the communication between sensors x and y if these two sensors happen to be deployed adjacent to one another. In particular, if sensors x and y become adjacent to one another, then these two sensors can use their shared symmetric key $K_{x,y}$ to authenticate one another (i.e. defend against impersonation) and to encrypt and decrypt their exchanged data messages (i.e. defend against eavesdropping).

It follows from this discussion that each sensor x in such a network is required to store $n-1$ symmetric keys, where n is the total number of sensors in the network and each stored key is shared between sensor x and a different sensor in the network. This requirement that each sensor in the network stores $n-1$ symmetric keys, where n is the number of sensors in the network, is rather severe especially when n is large and the available storage to store keys in every sensor is modest.

This situation raises the following important questions: Is it possible to design a keying protocol, where each sensor stores less than $n-1$ symmetric keys and yet the protocol is deterministically secure against impersonation, eavesdropping, and collusion?

In this Chapter, we show that the answer to this question is "Yes." In particular, we present a new keying protocol where each sensor stores only $(n+1)/2$ symmetric keys, and yet the protocol is deterministically secure against impersonation, eavesdropping, and collusion. We also show that this new protocol is near

optimal by showing that each sensor, in any keying protocol that is deterministically secure against impersonation, eavesdropping, and collusion, needs to store at least $(n-1)/2$ symmetric keys.

5.1 Sensor Networks and Adversaries

In this Chapter, we investigate a sensor network whose topology is not planned in advance, prior to the deployment of the network. Thus, when the network is deployed, any sensor can end up being adjacent to any other sensor in the network.

There are many occasions when a sensor network needs to be deployed before its topology can be planned in great detail. For example, when a wildfire breaks out unexpectedly, a sensor network that monitors the fire may need to be deployed in a hurry, before the network topology can be planned accurately. A second example, when a sensor network is deployed in a battlefield whose perimeter is continuously changing, the topology of the network cannot be determined fully until the time when the network is to be deployed. As a third example, if the deployed sensor network is mobile, then a detailed plan of the initial topology may be of little value.

In this network, when a sensor x is deployed, it first attempts to identify the identity of each sensor adjacent to x, then starts to exchange data with each of those adjacent sensors.

Any sensor z in this network can be an "adversary", and can attempt to disrupt the communication between any two legitimate sensors, say sensors x and y, by launching the following two attacks:

1. **Impersonation Attack:** Sensor z notices that it is adjacent to sensor x while sensor y is not. Thus, sensor z attempts to convince sensor x that it (z) is in fact sensor y. If sensor z succeeds, then sensor x may start to exchange data messages with sensor z, thinking that it is communicating with sensor y.

2. **Eavesdropping Attack:** Sensor z notices that it is adjacent to both sensors x and y, and that sensors x and y are adjacent to one another. Thus, when sensors x and y start to exchange data messages, sensor z can copy each exchanged data message between x and y.

To defend against these two types of attacks, sensors x and y need to share a symmetric key, denoted $K_{x,y}$ or $K_{y,x}$. The shared key $K_{x,y}$ needs to be known only to both sensors x and y, and not to any other sensor in the network, before these two sensors are deployed in the network. In Sections 5.3 and 5.4 below, we show how sensors x and y can use their shared key $K_{x,y}$ to defend against these two types of attacks.

5.2 Keying Protocols for Sensor Networks

A *keying protocol* for a sensor network is a scheme for assigning a unique symmetric key $K_{x,y}$ to each pair of distinct sensors x and y in the network. Each symmetric key $K_{x,y}$, that is assigned by the keying protocol, becomes known only to sensors x and y (and not to any other sensor in the network) before the network is deployed and before the adjacent sensors in the deployed network start to communicate with one another.

It follows from this discussion that if a sensor network has at most n sensors, then each sensor in the network needs to know at most $(n-1)$ distinct symmetric keys – one key $K_{x,y}$ for every other sensor y in the network – before the network is deployed.

There are two ways for a sensor x to know a symmetric key $K_{x,y}$ (before the network is deployed):

1. **Storage:**

   Key $K_{x,y}$ is stored in sensor x

2. **Computation:**

   Sensor x stores a constant kx that it can use to compute key $K_{x,y}$ as follows:

$$K_{x,y} := F(iy, kx)$$

   where

   F is a public function that is known to every sensor in the network

   iy is the identity of sensor y

   kx is a constant that is stored in sensor x

The *cost* of a keying protocol for a sensor network is measured by the number of symmetric keys (say $K_{x,y}$) plus the number of constants (say kx) that this keying protocol requires every x to store before the network is deployed.

Note that the cost of the straightforward keying protocol, which requires that every sensor x stores $(n-1)$ symmetric keys (of the form $K_{x,y}$), where n is the upper bound on the number of sensors in the network, is $(n-1)$.

In this Chapter, we address the following question. Is there a keying protocol for a sensor network, whose cost is much less than $(n-1)$, where n is the upper bound on the number of sensors in the network? Our research ends up with the following two results.

(a) **Efficiency:**

   There is a keying protocol, where each sensor shares a distinct symmetric key with every other sensor in the network, and whose cost is $(n+1)/2$.

(b) **Optimality:**

   The cost of every keying protocol, where each sensor shares a distinct symmetric key with every other sensor in the network, is at least $(n-1)/2$.

In the next Section, we present a keying protocol whose cost is $(n+1)/2$, which is half the cost of the straightforward keying protocol.

5.3 An Efficient Keying Protocol

Let n denote an upper bound on the number of sensors in our network. Without loss of generality, we assume that n is an odd positive integer. Each sensor in the network has a unique identifier in the range $0 \ldots n-1$. We use ix and iy to denote the identifiers of sensors x and y, respectively, in this network.

Two sensors, say sensors x and y, share a symmetric key denoted $K_{x,y}$ or $K_{y,x}$. Only the two sensors x and y know their shared key $K_{x,y}$. And if sensors x and y ever become neighbors in the network, then they can use their shared symmetric key $K_{x,y}$ to perform two functions:

1. **Mutual Authentication**: Sensor x authenticates sensor y, and sensor y authenticates sensor x.

2. **Confidential Data Exchange**: Encrypt and later decrypt all the exchanged data messages between x and y.

(Note that sensors x and y can become neighbors in the network in two cases. First, the two sensors x and y could be mobile and their movements cause them to become adjacent to one another. Second, the two sensors could be stationary and they are deployed adjacent to one another.)

In the remainder of this Section, we show that if the shared symmetric keys are designed to have a "special structure," then each sensor needs to store only $(n+1)/2$ shared symmetric keys. But before we present the special structure of the shared keys, we need to introduce two new concepts: universal keys and an asymmetric relation, named below, over the sensor identifiers.

Each sensor x in the network stores a symmetric key, called the *universal key* of sensor x. The universal key of sensor x, denoted ux, is known only to sensor x.

Let ix and iy be two distinct sensor identifiers. (Recall that both ix and iy are in the range $0 \ldots n-1$, where n is the (odd) upper bound of the number of sensors in the sensor network.) Identifier ix is said to be *below* identifier iy iff exactly one of the following two conditions holds:

1. $ix < iy$ and $(iy - ix) < n/2$
2. $ix > iy$ and $(ix - iy) > n/2$

The *below* relation is better explained by an example. Consider the case where $n = 5$. In this case, the sensor identifiers s are 0, 1, 2, 3, and 4, and we have:

- Identifier 0 is *below* identifiers 1 and 2.
- Identifier 1 is *below* identifiers 2 and 3.
- Identifier 2 is *below* identifiers 3 and 4.
- Identifier 3 is *below* identifiers 4 and 0.
- Identifier 4 is *below* identifiers 0 and 1.

The next three Theorems, concerning the *below* relation, are in order.

Theorem 1. *For any two distinct sensor identifiers ix and iy, one of the following two statements is true.*

1. *ix is below iy.*
2. *iy is below ix.*

Proof. Let ix and iy be any two distinct sensor identifiers. Thus, ix and iy are two distinct integers in the range $0 \ldots (n-1)$. Without loss of generality, assume that $ix < iy$. Because n is an odd integer, exactly one of the following two statements holds.

(1) $iy - ix < n/2$

(2) $iy - ix > n/2$

If statement (1) holds then ix is below iy. Otherwise statement (2) holds and iy is below ix. □

Theorem 2. *For each sensor identifier ix, the number of distinct sensor identifiers iy, where ix is below iy, is $(n-1)/2$.*

Proof. Each of the following $(n-1)/2$ sensor identifiers is below ix: $(ix-1) \mod n, (ix-2) \mod n, \ldots, (ix - \frac{n-1}{2}) \mod n$. Also, ix is below each of the following $(n-1)/2$ sensor identifiers: $(ix+1) \mod n, (ix+2) \mod n, \ldots, (ix + \frac{n-1}{2}) \mod n$. Thus, the number of distinct sensor identifiers iy, where iy is below ix, is $(n-1)/2$. Also, the number of distinct sensor identifiers iy, where ix is below iy, is $(n-1)/2$. □

Theorem 3. *For each sensor identifier ix, the number of distinct sensor identifiers iy, where iy is below ix, is $(n-1)/2$.*

Proof. The proof is similar to that of Theorem 2. □

The *special structure* of the symmetric key $K_{x,y}$, in the case where ix is below iy, is defined as follows:

$$K_{x,y} = H(ix|uy)$$

where

H is a secure hash function

$|$ is the concatenation operator

ix is the identifier of sensor x

uy is the universal key of sensor y

Note that in this case (where ix is below iy), the symmetric key $K_{x,y}$ needs to be stored in sensor x only since sensor y can compute this key (using H, $|$, ix, and uy) whenever it needs it.

Note also that in the other case, where iy is below ix, the special structure of the symmetric key $K_{x,y}$ is $H(iy|ux)$. And in this case, $K_{x,y}$ needs to be stored in sensor y only since sensor x can compute this key whenever it needs it.

The correctness of this keying protocol follows from the next Theorem.

Theorem 4. *If a sensor identifier ix is below a sensor identifier iy, then the symmetric key $K_{x,y} = H(ix|uy)$ is stored in sensor x and can be computed by sensor y when needed. No other sensor stores $K_{x,y}$ or can compute it.*

Proof. Assume that a sensor identifier ix is below a sensor identifier iy. By our keying protocol the symmetric key that is shared between sensors x and y, namely $H(ix|uy)$, is stored in sensor x only. Moreover, because sensor y is the only one that knows the universal key uy, only sensor y can compute the key $H(ix|uy)$. □

The efficiency of the keying protocol follows from the following Theorem.

Theorem 5. *Each sensor x stores one universal key ux and $(n-1)/2$ symmetric keys $K_{x,y}$ for every sensor y, where ix is below iy.*

Proof. According to the above keying protocol, each sensor x stores its universal key ux. Also, each sensor x stores the symmetric keys $K_{x,y}$ that sensor x shares with every sensor y where ix is below iy. From Theorem 2, there are $(n-1)/2$ sensors y where ix is below iy. Therefore, each sensor x stores $(n-1)/2$ symmetric keys. □

5.4 A Mutual Authentication Protocol

Before the sensors are deployed in a network, each sensor x is supplied with the following items:

1. One distinct identifier ix in the range $0 \ldots n-1$
2. One universal key ux
3. $(n-1)/2$ symmetric keys $K_{x,y} = H(ix|uy)$ each of which is shared between sensor x and another sensor y, where ix is below iy

After every sensor is supplied with these items, the sensors are deployed in random locations in the network.

Now if two sensors x and y happen to become adjacent to one another, then these two sensors need to execute a mutual authentication protocol so that sensor x proves to sensor y that it is indeed sensor x and sensor y proves to sensor x that it is indeed sensor y.

The *mutual authentication protocol* consists of the following six steps.

Step 1: Sensor x selects a random nonce nx and sends a hello message that is received by sensor y.

$$x \rightarrow y : hello(ix, nx)$$

Step 2: Sensor y selects a random nonce ny and sends a hello message that is received by sensor x.

$$x \leftarrow y : hello(iy, ny)$$

Step 3: Sensor x determines whether ix is below iy. Then it either fetches $K_{x,y}$ from its memory or computes it. Finally, sensor x sends a verify message to sensor

y.

$$x \rightarrow y : \mathit{verify}(ix, iy, H(ix|iy|ny|K_{x,y}))$$

Step 4: Sensor y determines whether iy is below ix. Then it either fetches $K_{x,y}$ from its memory or computes it. Finally, sensor y sends a verify message to sensor x.

$$x \leftarrow y : \mathit{verify}(iy, ix, H(iy|ix|nx|K_{x,y}))$$

Step 5: Sensor x computes $H(iy|ix|nx|K_{x,y})$ and compares it with the received $H(iy|ix|nx|K_{x,y})$. If they are equal, then x concludes that the sensor claiming to be sensor y is indeed sensor y. Otherwise, no conclusion can be reached.

Step 6: Sensor y computes $H(ix|iy|ny|K_{x,y})$ and compares it with the received $H(ix|iy|ny|K_{x,y})$. If they are equal, then y concludes that the sensor claiming to be sensor x is indeed sensor x. Otherwise, no conclusion can be reached.

Next, we describe how this mutual authentication protocol defends against two types of attacks, impersonation attacks and Man-in-the-Middle attacks.

1) Defending against Impersonation Attacks:

An impersonation attack by an adversary sensor z against the mutual authentication protocol can proceed as follows:

$$z \rightarrow y : \mathit{hello}(ix, nz)$$
$$z \leftarrow y : \mathit{hello}(iy, ny)$$
$$z \rightarrow y : \mathit{verify}(ix, iy, H(\ldots))$$

Note that in this attack, sensor z impersonates sensor x as it executes the mutual authentication protocol with sensor y. Fortunately, the computed term $H(\ldots)$ in the last verify message is incorrect for two reasons:

1. z does not know $K_{x,y}$.

2. z cannot replay an old verify message from x to y because the H in the replayed message does not have the correct nonce ny that was selected at random by sensor y in the second step of the protocol.

Thus, y cannot conclude that it is communicating with x and abandons the authentication protocol.

2) Defending against Man-in-the-Middle Attacks: A Man-in-the-Middle attack by an adversary sensor z against the mutual authentication protocol can proceed as follows:

$$
\begin{aligned}
x &\rightarrow z : \mathit{hello}(ix, nx) \\
z &\rightarrow y : \mathit{hello}(ix, nx) \\
z &\leftarrow y : \mathit{hello}(iy, ny) \\
x &\leftarrow z : \mathit{hello}(iy, ny) \\
x &\rightarrow z : \mathit{verify}(ix, iy, H(ix|iy|ny|K_{x,y})) \\
z &\rightarrow y : \mathit{verify}(ix, iy, H(ix|iy|ny|K_{x,y})) \\
z &\leftarrow y : \mathit{verify}(iy, ix, H(iy|ix|nx|K_{x,y})) \\
x &\leftarrow z : \mathit{verify}(iy, ix, H(iy|ix|nx|K_{x,y}))
\end{aligned}
$$

Note that in this attack, sensor z acts as a perfect medium relaying each message that it receives from sensor x to sensor y, and relaying each message that it receives from sensor y to sensor x. But in this case, the authentication protocol succeeds as it should, and the adversary sensor z does not gain any advantage by launching this attack.

5.5 A Data Exchange Protocol

After two adjacent sensors x and y have authenticated one another using the mutual authentication protocol described in the previous Section, sensors x and y can now start exchanging data messages according to the following *data exchange protocol*. (Recall that nx and ny are the two nonces that were selected at random by sensors x and y, respectively, in the mutual authentication protocol.)

Step 1: Sensor x concatenates the nonce ny with the text of the data message to be sent, encrypts the concatenation using the symmetric key $K_{x,y}$, and sends the result in a data message to sensor y.

$$x \rightarrow y : data(ix, iy, K_{x,y}(ny|text))$$

Step 2: Sensor y concatenates the nonce nx with the text of the data message to be sent, encrypts the concatenation using the symmetric key $K_{x,y}$, and sends the result in a data message to sensor x.

$$x \leftarrow y : data(iy, ix, K_{x,y}(nx|text))$$

Sensors x and y can repeat Steps 1 and 2 any number of times to exchange data between themselves.

Next, we describe how this data exchange protocol defends against two types

of attacks, eavesdropping attacks and replay attacks.

1) Defending against Eavesdropping Attacks:

An eavesdropping attack by an adversary sensor z against the data exchange protocol can proceed as follows:

$$x \rightarrow y, z : data(ix, iy, K_{x,y}(ny|text))$$
$$x, z \leftarrow y : data(iy, ix, K_{x,y}(nx|text))$$

In this attack, sensor z eavesdrops on the communication between x and y. However, sensor z cannot understand the text of the messages between x and y because this text is encrypted by using $K_{x,y}$ shared only between x and y.

2) Defending against Replay Attacks:

A replay attack by an adversary sensor z against the data exchange protocol can proceed as follows:

$$z \rightarrow y : data(ix, iy, K_{x,y}(ny'|text))$$
$$x \leftarrow z : data(iy, ix, K_{x,y}(nx'|text))$$

In this attack, the adversary sensor z waits until a new session, identified by the pair of nonces (nx, ny), is established between sensors x and y. Then sensor z starts to replay old data messages that were sent in an earlier session, identified by the pair of nonces (nx, ny), between x and y. But the replayed messages will be

discarded, and the attack will fail, because the intended receivers of the messages expect to find nx or ny, instead of nx' or ny', in these messages.

5.6 Optimality of Our Keying Protocol

According to our keying protocol, described in Section 5.3, each sensor in the network is required to store only $(n+1)/2$ keys. Thus, the total number of keys that need to be stored in the sensor network is $n(n+1)/2$. (This is much better than storing $n(n-1)$ keys in the sensor network as dictated by the straightforward keying protocol.)

Despite the big saving in storage, that is achieved by our keying protocol, one wonders "Is there another keying protocol that requires the network to store much less than $n(n+1)/2$ keys?" The following theorem indicates that the answer to this question is "No".

Theorem 6. *Each keying protocol, that is collusion-proof, requires the sensor network to store at least $n(n-1)/2$ keys.*

Proof. In order for a keying protocol to be collusion-proof, the sensor network needs to have $n(n-1)/2$ distinct symmetric keys. Thus, to prove that this theorem holds, it is sufficient to prove that every one of those symmetric keys, say $K_{x,y}$, causes a distinct key to be stored in sensor x or in sensor y. We carry out this proof by contradiction.

Assume that some symmetric key $K_{x,y}$ does not cause a distinct key to be stored either in sensor x or in sensor y. In this case, sensor x stores a key kx that x can use to compute at least two symmetric shared keys $K_{x,y}$ and $K_{w,x}$ as follows.

$$K_{x,y} = F(iy, kx) \tag{5.1}$$

$$K_{w,x} = F(iw, kx) \tag{5.2}$$

where F is a well-known function that can be used by each sensor to compute its shared keys from its stored keys.

Similarly, sensor y stores a key ky that y can use to compute at least two symmetric shared keys $K_{x,y}$ and $K_{y,z}$ as follows.

$$K_{x,y} = F(ix, ky) \tag{5.3}$$

$$K_{y,z} = F(iz, ky) \tag{5.4}$$

From 5.1 and 5.3 above, we have

$$F(iy, kx) = F(ix, ky) \tag{5.5}$$

Sensor x should not be allowed to utilize 5.5 and deduce key ky (in order that x be prevented from computing the shared key $K_{y,z}$). Therefore, there should not be any effectively computable function F', such that

$$F'(ix, F(ix, ky)) = ky \tag{5.6}$$

Similarly, sensor y should not be allowed to utilize 5.5 and deduce key kx (in

order that y be prevented from computing the shared key $K_{w,x}$). Therefore, there should not be any effectively computable function F'', such that

$$F''(iy, F(iy, kx)) = kx \tag{5.7}$$

From 5.6 and 5.7, we conclude the following.

(i) Because there is no effectively computable function F' that satisfies 5.6, there is no effective way to compute key ky in sensor y from key kx in sensor x before the two sensors x and y are deployed in the network.

(ii) Because there is no effectively computable function F'' that satisfies 5.7, there is no effective way to compute key kx in sensor x from key ky in sensor y before the two sensors x and y are deployed in the network.

From (i) and (ii), we conclude that the two secrets kx and ky cannot be computed and stored in sensors x and y respectively before these two sensors are deployed in the network. Contradiction! □

A keying protocol is called *uniform* iff this protocol requires each sensor in the network to store the same number of keys. Notice that the keying protocol described in Section 5.3 is uniform. Notice also that the next Theorem, concerning uniform keying protocols, follows from Theorem 6.

Theorem 7. *Each uniform keying protocol requires each sensor in the network to store at least* $(n-1)/2$ *keys.*

From Theorem 7, our keying protocol requires each process to store no more than one key beyond the number of keys that need to be stored in each process by the best uniform keying protocol. Thus, for all practical purposes, our protocol is the best uniform keying protocol for sensor networks.

5.7 Sensor Roles

There are two problems with our keying protocol, described in Section 5.3, of sensor networks.

1. This keying protocol requires that each sensor in the network stores $(n+1)/2$ symmetric keys, where n is an upper bound on the number of sensors in the network. Unfortunately, this number of symmetric keys (to be stored in each sensors), is still large in those cases when n is large. To make matters worse, we also showed that no keying protocol can require each sensor to store less than $(n+1)/2$ symmetric keys. (Luckily, this negative result depends on the fact that the architecture of the sensor network is as described in Section 5.1. And we show in this Section that if the network architecture is changed from that described in Section 5.1, one can design a keying protocol for the sensor network where each sensor is required to store much less than $(n+1)/2$ symmetric keys.)

2. If a sensor network, where each sensor stores $(n+1)/2$ keys, is deployed, and if later the number of sensors in the network to be increased beyond the upper bound n, the keys that are already stored in each deployed sensor need to be changed.

To solve these two problems, we introduce the concept of a "sensor role" as follows.

Each sensor in a sensor network has a role. The role of a sensor can describe the task that this sensor performs (e.g. sensing temperature or sensing motion). It can also describe the general location of the sensor (e.g. the second floor or the third floor of the building being sensed).

Many sensors in a sensor network can have the same role in order to provide fault-tolerance and accurate sensing. For example, a sensor network can have some

fifty sensors whose role is to sense temperature in the third floor of the building being sensed. These sensors can provide fault-tolerance and accurate sensing of the temperature in the third floor of the building.

If follows from this discussion that an upper bound m on the number of distinct roles in the network is relatively small whereas an upper bound n on the number of sensors in the same network is relatively large.

Now, the symmetric keys, in a sensor network whose sensors have roles, can be modified as follows. For every two roles f and g in the network, the network has a distinct symmetric key $K_{f,g}$. Only sensors, whose role is f or g, know $K_{f,g}$ and no other sensor in the network knows $K_{f,g}$.

To realize the symmetric keys $K_{f,g}$, we modify the above keying protocol as follows.

i. Each sensor whose role is f has an identifier if in the range $0 \ldots m-1$ where m is an upper bound on the number of distinct roles in the network. (This means that all sensors, whose role is f, have the same identifier if.)

ii. Each sensor whose role is f has a universal key uf (this means that all sensors, whose role is f, have the same universal key uf.)

iii. For any two roles f and g, the symmetric key $K_{f,g}$ is computed as follows (depending on whether if is below ig, or ig is below if, or $if = ig$).

Case 1. (if is below ig):

$$K_{f,g} = H(if|ug)$$

In this case, every sensor whose role is if stores $K_{f,g}$ and every sensor whose role is ig computes $K_{f,g}$.

Case 2. (ig is below if):

$$K_{f,g} = H(ig|uf)$$

In this case, every sensor whose role is ig stores $K_{f,g}$ and every sensor whose role is if computes $K_{f,g}$.

Case 3. $(if = ig)$:

$$K_{f,f} = H(if|uf)$$

In this case, every sensor whose role is if computes $K_{f,f}$.

Therefore, every sensor in the network, where sensors have roles, stores $(m+1)/2$ symmetric keys, where m is an upper bound on the number of roles in the network.

5.8 Related Work

There are two main keying protocols that were proposed in the past to reduce the number of stored keys in each sensor in the network. We refer to these two protocols as the probabilistic keying protocol and the grid keying protocol.

In the *probabilistic keying protocol* [33], each sensor in the network stores a small number of keys that are selected at random from a large set of keys. When two adjacent sensors need to exchange data messages, the two sensors identify which keys they have in common, then use a combination of their common keys as a symmetric key to encrypt and decrypt their exchanged data messages. Clearly, this protocol can probabilistically defend against eavesdropping.

Unfortunately, the probabilistic keying protocol suffers from the following problem. The stored keys in any sensor x are independent of the identity of sensor x and so these keys cannot be used to authenticate sensor x to any other sensor in the network. In other word, the probabilistic protocol cannot defend against impersonation.

In the *grid keying protocol* [44], [74], [3], and [31], each sensor is assigned an identifier which is the coordinates of a distinct node in a two-dimensional grid. Also

each symmetric key is assigned an identifier which is the coordinates of a distinct node in two-dimensional grid. Then a sensor x stores a symmetric key K iff the identifiers of x and K satisfy certain given relation. When two adjacent sensors need to exchange data messages, the two sensors identify which keys they have in common then use a combination of their common keys as a symmetric key to encrypt and decrypt their exchanged data messages.

The grid keying protocol has two advantages (over the probabilistic protocol). First, this protocol can defend against impersonation (unlike the probabilistic protocol) and can defend against eavesdropping (like the probabilistic protocol). Second, each sensor in this protocol needs to store only $O(\log n)$ symmetric keys, where n is an upper bound on the number of sensors in the network.

Unfortunately, it turns out that the grid keying protocol is vulnerable to collusion. Specifically, a small gang of adversarial sensors in the network can pool their stored keys together and use the pooled keys to decrypt all the exchanged data messages in the sensor network.

5.9 Concluding Remarks

Typically, each sensor in a sensor network with n sensors needs to store $n-1$ distinct (in order to ensure that the keying protocol is collusion-proof) shared symmetric keys to communicate securely with each other. Thus, the number of shared symmetric keys stored in the sensor network is $n(n-1)$. However, the optimal number of shared symmetric keys for secure communication, theoretically, is $\binom{n}{2} = n(n-1)/2$. Although there have been many approaches that attempt to reduce the number of shared symmetric keys, they lead to a loss of security: they are all vulnerable to collusion. In this Chapter, we show the best keying protocol for sensor networks, that needs to store only $(n+1)/2$ shared symmetric keys to each sensor. The number of shared symmetric keys stored in a sensor network with n sensors is $n(n+1)/2$, which

is close to the optimal number of shared symmetric keys for any key distribution scheme that is not vulnerable to collusion.

It may be noted that in addition to the low number of keys stored, and the ability to resist collusion between sensors, our keying protocol has two further advantages. First, it is uniform: we store the same number of keys in each sensor. Second, it is computationally cheap, and thus suitable for a low-power computer such as a sensor: when two sensors are adjacent to each other, the computation of a shared symmetric key requires only hashing, which is a cheap computation and can be done quickly. As our protocol has many desirable properties, such as efficiency, uniformity and security, we call this protocol the best keying protocol for sensor networks.

Chapter 6

ITY: Authentication in a Network without Identities

Most networks require that their users have "identities," i.e. have names that are fixed for a relatively long time, unique, and have been approved by a central authority (in order to guarantee their uniqueness). Unfortunately, this requirement, which was introduced to simplify the design of networks, has its own drawbacks. First, this requirement can lead to the loss of anonymity of communicating users. Second, it can allow the possibility of identity theft. Third, it can lead some users to trust other users who may not be trustworthy. In this Chapter, we argue that networks can be designed without user identities and their drawbacks. Our argument consists of providing answers to the following three questions. (1) How can one design a practical network where users do not have identities? (2) What does it mean for a user to authenticate another user in a network without identities? (3) How can one design a secure authentication protocol in a network without identities? We answered these three questions and published our results in [19].

Almost every network is designed under the assumption that each network user is assigned an *identity*, which is a name that satisfies three conditions:

i. *Fixed:* Once an identity is assigned to a network user, then this identity remains assigned to this user for a relatively long time, measured in months, years, or decades, even if this user decides to quit the network and no longer communicate with other users.

ii. *Unique:* The identity assigned to a network user is distinct from that assigned to any other network user.

iii. *Approved by a Central Authority:* The network has a central authority that generates or at least approves the identities assigned to all network users. This authority guarantees that (among other things) the identities assigned to distinct network users are distinct.

The first condition, *fixed identity*, needs some explanation. Assume that a user x in a network is assigned an identity id_x. Thus, when each other user in the network needs to send a message to user x, this other user needs to name id_x. Assume also that user x quits the network and its now available identity id_x is assigned to another user y in the network. Now if some user z, who is not yet aware that user x has left the network, decides to send a message to user x and names id_x, then the network delivers the sent message to the wrong user y (instead of discarding the message after recognizing that the intended message receiver has left the network). We conclude from this scenario that when a user leaves the network, its identity should be retired and not assigned to another user, at least for a relatively long time.

Some examples of user identities are as follows. The identity of a user phone in a phone network is the phone number that is assigned to this phone. The identity of a user computer in an IP network (in the Internet) is the IP address of this computer. Also, the identity of a user website in the World Wide Web is the URL assigned to this site.

The identities assigned to the users of a network play an important role in the execution of the network:

1. *User Identification:* When a user x wants to communicate with another user y, user x needs to supply the network with the identities of x and y so that the network can compute the best route for routing the exchanged messages between users x and y.

2. *User Authentication:* Any user x can be provided with a certificate that x can later use to prove to any other user that it is indeed user x. The certificate, provided to user x, has several items including the identity of user x and the public key of user x.

3. *User Reputation:* An identity is assigned to a network user for a relatively long time, and during this time, the reputation of this user, good or bad, can develop and take hold among other network users. Thus, each network can have "reputation systems" for recording and querying the reputations of network users. Note that these reputation systems cannot be developed unless the network users have unique and fixed identities.

Unfortunately, the adoption of user identities in a network does create some security holes in that network:

a. *Anonymity Loss:* Each message that is exchanged between users x and y needs to carry the identities of x and y in the clear in order to facilitate the routing of the message between x and y. Thus any user, that can observe this message while the message is in transit between x and y, can conclude correctly that users x and y are currently in communication (even if the message contents are encrypted).

b. *Identity Theft:* In communications that do not require strong user authentication, any user x, who happens to know the identity of another user y, can pretend to be user y while it communicates with a third user z.

c. *Misplaced Trust:* As mentioned above, the existence of user identities can facilitate the development of reputation systems. However, the data stored in some reputation systems can be corrupted, for example to indicate that some user x can be trusted whereas in fact user x is not trustworthy.

There are two approaches to address the security holes that are created by adopting user identities in a network. In the first approach, one develops techniques to defend against each one of these holes. For example, to defend against identity theft, one may require that each communication between any two users in the network should be preceded by strong mutual authentication.

In the second approach, one decides to design their network without (unique and fixed) user identities. In this case, the designed network will not have any of security holes that may be created by adopting user identifiers.

In this Chapter, we follow this second approach (simply because we believe that no one has attempted to follow this approach before), and attempt to answer the following three challenging questions:

i. How can one design a network without user identities?

ii. What does it mean for a user to authenticate another in such a network?

iii. How can one facilitate one user to authenticate another in such a network?

In the next Section, we answer the first question by outlining the architecture of a network that does not have user identifiers.

6.1 A Network without Identities

In this Section, we describe the architecture of a network where users do not have identities. In this network, instead of an identity, each user x has an *address*, denoted ad_x, and a nonempty set of *pseudonyms*, denoted NM_x. The value of the address ad_x and the contents of the set NM_x satisfy the following three conditions:

i. *Not Necessarily Fixed:* At any instant, each user x can change the value of its address ad_x or the contents of its pseudonym set NM_x.

ii. *Unique:* The value of ad_x for a user x is not equal to the value of ad_y for any other user y. Also, the contents of set NM_x for user x are disjoint from the contents of set NM_y for user y.

iii. *Approved by a Central Authority:* Only user x can request that the value of its address ad_x and the contents of its pseudonym set NM_x be changed. However, the network acts as a central authority, and declines any part of the request that violates the above *uniqueness* condition. For example, if user x requests that the value of its address ad_x be changed to a value that is currently being claimed for address ad_y for another user y, then the network will decline the request of user x.

Notice that the first condition, that ad_x and NM_x are required to satisfy, is the opposite of the first condition that an identity of user x is required to satisfy. This shows that ad_x and NM_x do not constitute an identity of user x.

With ad_x and NM_x, we design the three protocols: (1) registration protocol (2) connection protocol (3) authentication protocol. In the registration protocol, each user sends a registration message to the network every T seconds. In the connection protocol, a user x sends a request message to the network requesting to be connected to another user y, and the network replies by sending reply messages to both x and y informing them that they have been connected. In the authentication

protocol, two connected users exchange and verify their tokens in order to check whether they had communicated earlier. We discuss these three protocols in detail in the following three Sections.

The value of address ad_x indicates a physical location where user x can receive messages. Thus when some user y wants to send a message to user x, user y sends the message to ad_x. Each pseudonym nm_x in the pseudonym set NM_x of user x is meant to identify user x in one connection with another user in the network.

Because at any time each user x can update the value of its address ad_x and the contents of its pseudonym set NM_x, user x needs to register in the network, every T seconds, the current value of ad_x and the current contents of NM_x. Thus, every T seconds, user x sends to the network a registration message that contains the current value of ad_x and the current contents of NM_x. The network maintains a registration table where it stores the latest registered address ad_x and the latest registered pseudonym set NM_x for each user x in the network.

When a user x with pseudonym nm_x wants to communicate with another user y with a pseudonym nm_y, user x sends a request message, that contains ad_x, nm_x, and nm_y to the network. Then the network searches in its registration table for a user y with pseudonym nm_y. If the network finds no such user y in the registration table, the network rejects the request. If the network finds (exactly) one such user in the registration table, the network connects x to this user y.

Next, the network computes a symmetric connection key CK and sends reply messages to both users x and y. The reply message to user x contains ad_x, nm_x, ad_y, nm_y, and the connection key CK. The reply message to user y contains ad_x, nm_x, ad_y, nm_y, and CK. When users x and y receive their respective reply messages from the network, they can start exchanging messages that are encrypted using the connection key CK.

Once users x and y receive their respective reply messages from the network and recognize that they are connected, they proceed to execute an authentication protocol in order that each of them authenticates the other. But what does it mean for a user to authenticate another in this network (where users have no identities)? We answer this question in the next Section.

6.2 User Authentication in the Network

Consider the case where a user x with a pseudonym nm_x was connected to (and communicated with) another user y with a pseudonym nm_y as many as k times, where k is at least one. Later, user x with its pseudonym nm_x requests from the network to be connected, for the $(k+1)$-th time, to a user with the pseudonym nm_y and the network grants user x its request. Now how can either user (x or y, respectively) be sure that it is connected to the same user (y or x, respectively) to whom it was connected k times in the past?

This is not an easy question to answer. For example, it is possible that after users x and y were connected k times in the past, user y gave up its pseudonym nm_y and a third user z later claimed nm_y as one of its pseudonyms. Now, when user x requests to be connected to a user with the pseudonym nm_y, user x is connected to user z instead of user y.

The answer to the above question is a new authentication protocol that we designed for our network. When two users x and y are connected by the network, if either of these two users, say user x, thinks that it had been connected to the other user, user y, several times in the past, then executing the authentication protocol by the two users x and y, can lead user x to know for sure whether the other user, user y, is the same user to whom user x was connected several times in the past.

Our design of the authentication protocol is intended to defend against an adversary that can perform two dangerous operations:

i. *Eavesdropping:* The adversary can read every message that is sent between any user and the network or between any two connected users in the network. In particular, the adversary can read every registration message and can compute and maintain an accurate copy of the registration table that is stored in the network. Note that the exchanged messages between (connected) users are encrypted using connection keys and so the adversary cannot *understand* them, even if it does read them.

ii. *Impersonation:* The adversary can pretend to be a user in the network and send a message to the network or to any other user in the network. The adversary can also pretend to be the network and send a message to any user. Each message, that is sent by the adversary, is composed using the knowledge that the adversary has gained from reading all the sent messages in the network. For example, the adversary can "replay" a message that has been sent earlier in the network.

    Note that the adversary cannot impersonate the network, because we assume that (1) the network has a private key whose corresponding public key is known to all users in the network, and that (2) the network uses its private key to sign every (reply) message that the network sends to a user.

The objective of the adversary is to be connected to a user x in the network and then to use the authentication protocol to convince user x that it (the adversary) is the same user y to whom user x was connected several times in the past.

The designed authentication protocol is simple enough. When two users x and y are connected, each of the two users selects a new pseudonym and a new "token". Then the two users exchange their new pseudonyms and new tokens encrypted using the connection key CK. Let nm_x and tk_x be the new pseudonym and new token selected by user x and let nm_y and tk_y be the new pseudonym and new token selected by user y. After exchanging their new pseudonyms and tokens,

the two users x and y end up with the following tuple which defines their next authenticated connection:

$$[nm_x, tk_x, nm_y, tk_y]$$

Now assume that user x wants to establish the next connection to user y, and then x initiates the connection protocol indicating that its pseudonym is nm_x and that it wants to be connected to a user with the pseudonym nm_y. There are two cases that need to be considered in this scenario.

In the first case, the network connects user x with the correct user y where both x and y have the same two tokens. In this case, the authentication protocol proceeds as follows: user x sends tk_x to user y which checks that the received token is the expected one and sends in turn tk_y to user x which checks that the received token is the expected one.

In the second case, the network connects user x with a user z, different from the correct user y. (This could have happened as follows. First, user y decided to give up its pseudonym nm_y, then later user z decided to claim nm_y as one of its pseudonyms. Thus when user x requested to be connected with the pseudonym nm_y, the network connects user x to user z which does not know either of the two tokens tk_x and tk_y.)

In this second case, the authentication protocol proceeds as follows. User x sends tk_x to user z which sends back an arbitrary value (different from tk_y) to user x which recognizes that it is communicating with a different user than user y. Thus, each of the two users concludes that it is communicating with the other user for the first time.

In either case, at the end of the authentication protocol, user x selects a new pseudonym nm'_x and a new token tk'_x and sends them to the other user, whether y or z. Also, the other user, whether y or z, selects a new pseudonym nm'_y and a new token tk'_y to user x. Thus both user x and the other user, whether y or z, end up

with the following tuple which defines their next authenticated connection:

$$[nm'_x, tk'_x, nm'_y, tk'_y]$$

So far, we outlined broadly the three protocols in our network (where users have no identities): the registration protocol, the connection protocol, and the authentication protocol. In the registration protocol, each user sends a registration message to the network every T seconds. In the connection protocol, a user x sends a request message to the network requesting to be connected to another user y, and the network replies by sending reply messages to both x and y informing them that they have been connected. In the authentication protocol, two connected users exchange and verify their tokens in order to check whether they had communicated earlier.

In the next three Sections, we discuss these three protocols in great detail.

6.3 Registration Protocol

The function of the registration protocol is to allow each user x in the network to periodically register its current address ad_x and its current pseudonym set NM_x. This protocol also allows each user to periodically register its current registration key RK_x, which is a public key, selected at random by user x, whose corresponding private key is known only to user x.

The registration protocol requires that, every T seconds, each user x sends to the network a *registration message* of the following form: $(ad_x, NM_x, RK_x, t_x, sign_x)$ where ad_x is the current address of some user, and NM_x is the current pseudonym set of the user at ad_x, and RK_x is the current registration key of the user at ad_x, and t_x is the real time, or timestamp, of the user at ad_x when this user sends the registration message, and $sign_x$ is the message signature signed by the private key that corresponds to RK_x.

Table 6.1: Registration Table

address	pseudonym set	registration key	timestamp
ad_x	NM_x	RK_x	t_x
ad_y	NM_y	RK_y	t_y

The network stores the data, that are contained in the received registration messages into a table called the *registration table*. The registration table has four columns, also called attributes, named address, pseudonym set, registration key, and timestamp. The index attribute of this table is the address. Table 6.1 illustrates the registration table when it has two tuples.

When the network receives a registration message $(ad_x, NM_x, RK_x, t_x, sign_x)$ from a user at address ad_x, the network updates the registration table by executing the following protocol:

Step 1: If the timestamp t_x in the message is not "close" to the real time of the network or if the message signature $sign_x$ is not correct, then the network discards the message and terminates the protocol.

Step 2: If the network finds no tuple in the registration table whose address is ad_x, then the network adds the tuple $[ad_x, NN_x, RK_x, t_x]$ to the registration table, where NN_x is the same set as NM_x after removing from it every pseudonym that already occurs in the registration table, and terminates the protocol.

Step 3: If the network finds a tuple $[ad'_x, N'_x, RK'_x, t'_x]$ in the registration table where $ad_x = ad'_x$ and $RK_x = RK'_x$, then the network replaces this tuple by the tuple $[ad_x, NN_x, RK_x, t_x]$ in the registration table, where NN_x is the same set as NM_x after removing from it every pseudonym that already occurs in the registration table, and terminates the protocol.

Periodically, the network checks the registration table and discards every tuple that has not been updated for more than $2T$ seconds. Note that there are three causes for a tuple $[ad_x, NM_x, RK_x, t_x]$ in the registration table not to be updated for more than $2T$ seconds:

i. User x has failed or has quit the network.

ii. User x has changed its address from ad_x to ad'_x (possibly because user x has "moved" from one location to another).

iii. User x has changed its registration key from RK_x to RK'_x (possibly to prevent its fixed registration key RK_x from becoming a fixed identity of user x).

If user x fails or leaves the network, then its tuple in the registration table remains unchanged for more than $2T$ seconds. This causes the network to remove this tuple from the registration table.

If user x changes its address from ad_x to ad'_x, then the first registration message after the change will cause a new tuple (that has the new address ad'_x) to be added to the registration table, leaving the old tuple (that has the old address ad_x) unchanged. The old tuple remains unchanged in the registration table for more than $2T$ seconds, causing the network to remove this old tuple from the registration table.

If user x changes its registration key from RK_x to RK'_x, then the registration messages after this change will be discarded, leaving the tuple of user x in the registration table unchanged. This tuple of user x (with the old registration key RK_x) remains unchanged in the registration table for more than $2T$ seconds, causing the network to remove this tuple from the registration table. Once this tuple is removed, the next registration message from user x will cause a new tuple of user x (with a new registration key RK'_x) to be added to the registration table.

6.4 Connection Protocol

The function of the connection protocol is to allow two users in the network to become *connected* to one another. This means that

i. each of the two users knows the current address of the other user (and so the two users can now exchange messages), and

ii. the two users share a symmetric key, called their connection key CK, that they can use to encrypt and decrypt their exchanged messages.

The connection protocol consists of three messages: a *request message* from any user x to the network requesting that user x be connected to another user y followed by two *reply messages* from the network to the two users x and y informing them that they have been connected.

When a user x with a pseudonym nm_x wants to establish a connection with another user y with a pseudonym nm_y, user x sends to the network a request message of the form: $(ad_x, nm_x, nm_y, t_x, sign_x)$ where ad_x is the currently registered address of user x, and nm_x is a currently registered pseudonym of user x, and nm_y is a currently registered pseudonym of user y, and t_x is the real time, or timestamp, of user x when it sent the request message, and $sign_x$ is the message signature signed by the private key that corresponds to the current registration key RK_x of user x.

When the network receives a connection request message $(ad_x, nm_x, nm_y, t_x, sign_x)$, it executes the following protocol:

Step 1: If the timestamp t_x in the request message is not "close" to the real time of the network, or if the network finds no tuple in the registration table whose address is ad_x, then the network discards the message and terminates the protocol.

Table 6.2: Authentication Table of User x

my-pseudonym	my-token	other-pseudonym	other-token
nm_x	tk_x	nm_y	tk_y

Table 6.3: Authentication Table of User y

my-pseudonym	my-token	other-pseudonym	other-token
nm_y	tk_y	nm_x	tk_x

Step 2: If the network finds in the registration table two distinct tuples, $[ad'_x, NM'_x, RK'_x, t'_x]$ and $[ad'_y, NM'_y, RK'_y, t'_y]$ where $ad_x = ad'_x$, and $nm_x \in NM'_x$, and $nm_y \in NM'_y$

then the network does the following:

- it selects at random a symmetric connection key CK.
- it sends a reply message of the form $(ad_x, nm_x, ad'_y, nm_y, t_x, \{CK\}_{RK'_x}, sign_N)$ to ad'_x.
- it sends a reply message of the form $(ad_x, nm_x, ad'_y, nm_y, t_x, \{CK\}_{RK'_y}, sign_N)$ to ad'_y.

where $sign_N$ is the message signature signed by the private key of the network, whose corresponding public key is known to all users in network.

Otherwise, the network discards the message and terminates the protocol.

6.5 Authentication Protocol

When a user x wants to communicate with another user y, user x initiates the connection protocol, presented in Section 6.4, in order to achieve two goals:

i. Each of the two users obtains the current address of the other user and so the two users can start to exchange messages.

ii. Each of the two users obtains a copy of the symmetric connection key CK, and so the two users can encrypt and decrypt all their exchanged messages.

After the connection between users x and y is established, and before x and y can start exchanging data messages over the established connection, users x and y need to execute the authentication protocol in order that each of them can determine whether or not the established connection is the "first" connection between x and y.

Consider the case where this established connection is not the first one between users x and y. In this case, users x and y have agreed on four items in their last established connection:

1. nm_x: is a new pseudonym for user x.
2. nm_y: is a new pseudonym for user y.
3. tk_x: is a new token for user x.
4. tk_y: is a new token for user y.

Moreover, each of the two users has stored these agreed-on four items in a local table, called the *authentication table*, of the user. Table 6.2 and Table 6.3 show the authentication table of user x and user y, respectively.

Note that each authentication table has four attributes named: my pseudonym, my token, other pseudonym, and other token.

Thus, in this case, before user x sent a request message to initiate the current connection to user y, user x needed to consult its authentication table in order to determine its own pseudonym and the pseudonym of user y that needed to be included in the request message.

The authentication protocol between user x, with pseudonym nm_x, and user y, with pseudonym nm_y, proceeds in seven steps as follows:

Step 1: If user x finds in its authentication table a tuple of the form: $[nm_x, tk_x, nm_y, tk_y]$, then user x assigns to its boolean flag $conn_x$ the value true. Otherwise, user x assigns to its flag $conn_x$ the value false.

Step 2: User x sends the message $\{u\}_{CK}$ to ad_y where the value of u depends on the value of $conn_x$ as follows. If $conn_x$ is true, then u is the token tk_x in the above tuple. Otherwise, u is selected at random by user x.

Step 3: When user y receives $\{u\}_{CK}$ from ad_x, then user y sends $\{v\}_{CK}$ to ad_x, where v is computed as follows. If user y finds in its authentication table a tuple of the form $[nm_y, tk_y, nm_x, u]$, then v is the token tk_y in the tuple, and user y assigns its flag $conn_y$ the value true. Otherwise, v is selected at random by user y, and user y assigns its flag $conn_y$ the value false.

Step 4: When user x receives $\{v\}_{CK}$ from ad_y, then user x computes the value of its flag $conn_x$ as follows. If user x finds in its authentication table a tuple of the form: $[nm_x, tk_x, nm_y, v]$, then user x assigns its flag $conn_x$ the value true. Otherwise, user x assigns $conn_x$ the value false.

Step 5: User x sends $\{nm'_x, tk'_x\}_{CK}$ to ad_y, where nm'_x and tk'_x are a new pseudonym and token selected at random by user x. Then, user y sends $\{nm'_y, tk'_y\}_{CK}$ to ad_x, where nm'_y and tk'_y are a new pseudonym and token selected at random by user y.

Step 6: If flag $conn_x$ is true, then user x removes the tuple $[nm_x, tk_x, nm_y, tk_y]$ from its authentication table. And, in any case, user x adds the tuple $[nm'_x, tk'_x, nm'_y, tk'_y]$ to its authentication table.

Also, if flag $conn_y$ is true, then user y removes the tuple $[nm_y, tk_y, nm_x, tk_x]$ from its authentication table. And, in any case, user y adds the tuple $[nm'_y, tk'_y, nm'_x, tk'_x]$ to its authentication table.

Step 7: If the two flags $conn_x$ and $conn_y$ are both true, then each of the two connected users x and y is sure that the other user is the same one to which it was connected in the past.

Otherwise, the two flags $conn_x$ and $conn_y$ are both false and each of the two users x and y is sure that the other user is a new one to which it was not connected in the past.

After executing the above authentication protocol, the two users x and y can now start to exchange data messages encrypted using the connection key CK.

At the end of the authentication protocol, user x has a new tuple $[nm'_x, tk'_x, nm'_y, tk'_y]$ in its authentication table, and user y has a corresponding tuple $[nm'_y, tk'_y, nm'_x, tk'_x]$ in its authentication table. As long as these two tuples remain in their respective authentication tables, the two users x and y can authenticate one another correctly in the next time they become connected in the future. However, it is possible that one of these two users, say user x, may decide to discard the tuple $[nm'_x, tk'_x, nm'_y, tk'_y]$ from its authentication table. In this case, the next time users x and y become connected, their execution of the authentication protocol will indicate (incorrectly) that users x and y are being connected for the first time.

Note that whenever a user x decides to drop one of its pseudonyms nm_x from its pseudonym set NM_x, then user x should also drop from its authentication table any tuple where the my-pseudonym attribute has the value nm_x.

6.6 Protocol Security

In the previous Sections, we have described the working of a network without identities. Our network has three protocols – the registration protocol, the connection protocol, and the authentication protocol. Our description shows that, even if there are no permanent identities, a user x can find another user (say y, though x does not

know the identity of y) by searching for his pseudonym (say nm_y). In this Section, we focus on another important aspect of the network, its security.

In this Section, we assume two users x and y are communicating using our network. The adversary is another user z. The powers of the adversary are as listed in Section 6.2.

We now demonstrate that our three protocols are designed to defend against many attacks that could potentially compromise the security of the network. We provide seven example attacks that can be launched by adversary z, and show how our protocols defeat these attacks.

Pseudonym Theft Attack

In this attack, adversary z seeks to acquire pseudonym nm_y, which is currently held by user y. The network, which maintains the registration table, does not allow two users to simultaneously register the same pseudonym. Hence, as long as nm_y is registered to y, z will fail to acquire nm_y.

We note that the registration protocol is soft-state: if a time period of $2T$ passes, and y does not send a message to update its tuple in the registration table, then the entire tuple is discarded. z can now acquire the pseudonym nm_y. But z cannot cause this to happen (unless y voluntarily leaves the network), because z cannot delete registration messages sent by y to the network.

Address Theft Attack

This attack is similar to the one above. z seeks to acquire address ad_y, currently held by user y. We can show the network is secure against this attack by adapting our argument for the case of pseudonym theft. The network only allows one user to be present at an address, so z cannot capture an address that is already registered. z can send registration messages to claim the address – but as z does not have the

private key corresponding to RK_y, these messages have at least one of the following two characteristics:

1. Wrong registration key. The message does not contain RK_y.
2. Not properly signed. The message contains key RK_y, but is not signed by the private key corresponding to RK_y.

In both these cases, the registration protocol simply discards the messages. Also, z cannot prevent y from updating its registration tuple. Hence z cannot steal an address from y, while y is in the network.

Wrong Address Attack

In this attack, z tries to acquire an address ad'_z that is not its true address ad_z. Given that ad'_z is not the address of any other user, z succeeds in obtaining this address. But this attack is of no practical value – z successfully acquires a new address, but cannot send messages from it or receive messages sent to it (because it is not physically present at this address). Hence, even though this attack succeeds, it is of no importance.

Message Forging Attack

In this attack, z sends messages, either to the network or to a user (say x), which purport to be from user y. We have demonstrated, in our discussion of the address theft attack above, that z cannot successfully forge registration protocol messages.

In the case of connection protocol messages, note that every message is signed by the sender's private key. Consider a message that claims to be from a user with pseudonym nm_y, where nm_y is in fact a pseudonym of user y (and not user z). The network can verify the signature in the message (because, by inspecting the registration table, the network can find the public key to check the signature: it is

the registration key of the user with pseudonym nm_y). z does not have the private key of y, and therefore cannot forge connection protocol messages to look like they are from y. Similarly, z does not have the private key of the network, and cannot forge connection protocol messages to look like they were sent by the network.

All the messages in the authentication protocol are encrypted by a shared key CK, which is not known by z: CK is randomly chosen by the network, and is sent to the users x and y, encrypted with their respective registration keys. Consequently, z cannot even read, much less write messages encrypted by CK. Hence, we conclude that z cannot successfully forge messages in any of our three protocols.

Replay Attack

In the replay attack, the adversary makes additional copies of a legitimate message, which was originally sent by another user (or by the network), and sends the copies to the destination of the original. In other words, z causes multiple copies (rather than a single copy) of a message to arrive at its destination.

In order to solve the problem of replay attacks, we design the messages in our protocols to bear timestamps. If the recipient receives a message which is delayed, it discards the message as being stale.

Impersonation Attack

In the impersonation attack, z tries to convince a user x that he (z) is in fact another user y, known to x as nm_y. Our connection protocol ensures that, when a user x believes that he is connected to a user with pseudonym nm_y, this is in fact true. It is therefore essential for z to register pseudonym nm_y. As long as y is present in the system, and has pseudonym nm_y, this is also impossible: the network does not allow multiple users to share the same pseudonym. But it is possible for y to leave the system, in which case z can acquire the pseudonym nm_y.

Suppose z does acquire nm_y, and obtains a connection to nm_x. z still fails to impersonate y, because, in addition to the use of pseudonyms, x and y also use a pair of tokens to authenticate themselves to each other. z does not know tk_y, the token used by y (the chance of guessing tk_y correctly is vanishingly small). Hence, z cannot supply tk_y, as required by the authentication protocol, and x can identify z as being distinct from y.

Is it possible for z to know tk_y? Note that, if instead of initiating the connection, z simply waits for x to connect to nm_y, then in accordance with the authentication protocol x supplies its own token tk_x. It is natural to question whether z can leverage this feature to obtain tk_y.

Unfortunately for adversary z, this is not possible. The only user who can send tk_y, is y, who has already left the network. To counter this argument, suppose we consider a different scenario, where z obtains tk_y before y leaves the network. Even in this case, this attack is not possible. y will only send tk_y to nm_x. The only way z can obtain tk_y is by waiting for x to leave the network, then acquiring nm_x and waiting for y to contact nm_x. But in this case, x has already left the network, so obtaining tk_y is useless: there is no longer a party who can be deceived using token tk_y. (Note that, if x does rejoin the network later, he starts with a new pseudonym and a fresh authentication table; token tk_y no longer authenticates y to x.)

Man-In-The-Middle Attack

In the Man-in-the-middle or Janus attack, adversary z simultaneously impersonates x to y and y to x. x and y believe that their conversation is private, but in fact all messages are seen (in the clear) by z.

Our analysis of this attack follows directly from that of the impersonation attack. In order to impersonate x to y, z must have the pseudonym nm_x; to impersonate y to x, he must have nm_y. If in fact, he manages to acquire both pseudonyms,

nm_x and nm_y, then this shows that x and y have left the network. There is no user in the network who can be affected by the attack. Hence, our network is robust against Man-in-the-middle attacks.

6.7 Related Work

The problem of anonymous communication over a network is an old and respected problem, and has inspired a considerable amount of research. The original papers proposing architectures for anonymous networks were MIX-net [15], DC-net [16], and Crowds [99]. MIX-net, the original anonymous communication system, provided untraceable email by making use of public key cryptography to hide the participants and contents of communications. Email only requires periodic deliveries, and DC-net was proposed for applications requiring continual delivery over unconditionally secure channels. Crowds, an application-level protocol, uses probabilistic random forwarding. Unfortunately, it has scalability problems, as it depends on a centralized admission control server. Most subsequent solutions have implemented variants of the same basic idea: to have nodes relaying traffic, so it is hard to determine where a message originated. Perhaps the best known, Tor [28] attempts to provide a variant of Onion Routing [112]. Tor uses directory servers that maintain router information. A user can send (encrypted) traffic on a long path through many proxy servers, before a Tor node finally sends the (unencrypted) message to its final destination. Unfortunately, Tor has well-known scaling problems.

In order to resolve the scalability problem of Tor, some authors propose peer-to-peer networks to relay messages. One such solution, APFS [109], provides two variants: 1) unicast communication with a central coordinator, and 2) multicast routing. Another solution by Tarzan [38], a P2P anonymous IP network overlay, extends MIX-net using ideas very similar to Tor: layered encryption and multihop routing. Further, Tarzan uses gossip-based protocols for peer discovery, rather than

a central directory. Despite all these scalability improvements, Tarzan still has problems scaling to large networks.

Distributed hash tables (DHTs) are often used to try and overcome the scalability and security problems of the P2P-based approach. For instance, Salsa [88] is a structured approach to organizing highly distributed anonymous communications systems. Salsa uses its own secure lookup operation over a custom DHT to locate forwarder nodes. Cashmere [129] focuses on the fragility of paths through a network when there is anonymous communication. Its solution is to use regions in the overlay name space as mixes, rather than single nodes; this reduces the probability of a mix failure. AP3 [82], a cooperative decentralized anonymous communication service, is similar to Crowds: it selects random relays, and implements routing dynamically.

Despite this extensive body of research, the problem of constructing a truly secure anonymous network is still open. Mittal and Borisov [83] demonstrate how to compromise Salsa and AP3 with information leaks. More recently, Tran et al. have discovered information leaks in Salsa and Cashmere, Hintz in Safeweb using SSL [54], and Ristenpart et al. in cloud computing-based anonymous networks [101]. There continues to be a great deal of research into the problem of making it hard to trace the IP address of the initiator who originates traffic.

However, a characteristic feature of all the above authors is that they universally assume that IP address is identity. The aim of these networks is to obfuscate the link between the user of a network, or more precisely his role on the network, and his IP address, i.e. his identity. The above authors do not consider exactly what constitutes an identity, and what it means to protect the identity of a user. This is a glaring shortcoming, considering the critical importance of the concept of identity: not only is the word itself ubiquitous in security (for example, consider the Department of Homeland Security's recent draft, "The National Strategy for Trusted Identities in Cyberspace" [1]), it is the basis of many concepts such as

access policy, information integrity protocols, and reputation systems.

In this Chapter, we attempt to define the concept of 'identity', and suggest that an identity is a persistent, unique, and undisputed (hence distributed by a central authority) name. Having provided this definition, we then ask the very important question, "Is it possible to have a network without identities?" The question seems absurd at first glance - how is it possible to trust some users, and give them exclusive rights and privileges (not enjoyed by other users), without identities? But we show that, in fact, it is indeed possible to build a network where users authenticate themselves to each other using only their pseudonyms – "temporary names" adopted for use on the network – without the use of fixed identities. Further, we demonstrate that this network is highly resistant to a wide variety of attacks. We suggest that this may in fact be an example of a novel class of networks.

6.8 Concluding Remarks

The usual structure of networks, where users are assigned unique identities, makes communication between users – as well as development of relationships between them – simple. But this simple structure comes at a price. Clearly, associating an identity with a user leads to loss of anonymity; there may be concerns about reputation – other users can judge one's actions; and the network itself may show biased behavior. Most importantly, there exist attacks which seek to steal a user's identity.

In this Chapter, we present the outline of a network in which users do not have identities. Users are contacted by searching for their "pseudonyms", which they change frequently. Authentication is done by users themselves, not by the certification of a central authority. In this network, as there is no identity, there is no identity theft. Further, we show in Section 6.6 that the network is robust against many different kinds of attacks, notably the impersonation and Man-in-the-middle

attacks.

Besides the theoretical novelty of the idea, we are pleased to report that it shows considerable promise for future research. The entire concept of a network without identities is very interesting, as it opens up the question of inter-user relationships without external reputations; indeed, we venture to suggest that this may be a whole new kind of network, distinct from both traditional client-server and reputation-based peer-to-peer networks. In our own immediate future work, we are attempting to develop our network in more detail, so that it becomes robust against other attacks such as denial-of-service and message loss.

Chapter 7

Concluding Remarks

In this dissertation, we proposed five authentication protocols to address five critical problems related to authentication over computer networks.

In Chapter 2, we observed that HTTP suffers from many prevalent attacks – server impersonation, message modification, and cookie theft and injection attacks. This observation motivated us to design HTTPI, an HTTP with Integrity. HTTPI is lightweight like HTTP and compatible with middle boxes like caching proxies and yet provides some security guarantees – server authentication, message integrity, and cookie integrity.

In Chapter 3, we observed that HTTPS is vulnerable to human mistakes and Phishing and Pharming attacks. Based on these observations, we designed TLP to replace TLS and the standard password protocol. TLP provides a new class of brown pages and a new login protocol to defend against these attacks.

In Chapter 4, we argued that the usual combination of TLS and the standard password protocol is vulnerable to a variety of Phishing attacks, and proposed to replace the standard password protocol with the Two-Way Password Protocol (TPP). TLS is vulnerable to Phishing attacks because clients do not have information to decide whether the displayed web site is what users intended. In order

to defend against a variety of Phishing attacks, we designed TPP to complement the standard password protocol with TLS. TPP is improved over TLP since TPP is more lightweight than TLP by reducing the number of messages. In addition to that, TPP is more compatible with the current system, TLS with the standard protocol than TLP by replacing only the standard password protocol.

In Chapter 5, we observed that every sensor in a sensor network is required to store $n - 1$ symmetric keys for secure communication if there are n sensors. Considering the storage constraints of sensor networks, it is important to reduce the number of key and support secure communications. Existing keying protocols focused on the reduction of keys and they are still vulnerable to collusion attacks. We designed Best Keying Protocol (BKP) to require every sensor to store only $(n + 1)/2$ and to defend against collusion attacks.

In Chapter 6, we argued that most networks are designed with "identities" because identities facilitate the working of networks. However, identities create security problems – anonymity loss, identity theft, and misplaced trust. There can be two approaches to solve these problems. One is to defend against every attack related with identities and the other one is to design a network without identities. We took the second approach to design a network without identities because it has not previously been attempted. We found that addresses and pseudonyms cannot be used for identities because they are not fixed. With addresses and pseudonyms, we designed anonymous authentication for a network without identities.

These five protocols make a suite of authentication protocols. This suite of authentication protocols can be used to authenticate clients and servers for a mutual authentication depending on applications. Secure applications like bank web sites requiring strong mutual authentications can adopt TLP or TPP with TLS. Most web sites not requiring confidentiality can use HTTPI to defend against impersonation, message modification, and cookie theft and injection attacks. BKP can be used for

a mutual authentication under the constraints of resources such as sensor networks. Lastly, ITY can be used to authenticate anonymous communications.

7.1 Future Research

In this dissertation, we have presented five authentication protocols that can be deployed in various environments of computer networks including the World Wide Web, sensor networks, and anonymous networks. The ideas used in designing these five authentication protocols can be also used in designing new authentication protocols in other environments:

1. Virtualization Environments
2. Delegation Environments
3. Cloud Computing Environments

Next, we discuss the new challenges of authentication in these three environments.
Virtualization Environments: Virtualization has become the norm in enterprise computing and is also widely used by individuals. Virtualization enables a separate identity to each virtual machine and if a computer has multiple virtual machines, the computer can assume the multiple identities of these machines. In this case, virtual machines might need to communicate with one another and authentication among multiple virtual machines needs to be considered. If authentication does not exist in this environment, any malicious virtual machine can eavesdrop and impersonate to any other virtual machines and can subvert this system. Thus, an effective authentication protocol for virtual environments needs to be designed.
Delegation Environments: The World Wide Web has seen a different model other than the client-server model. In client-server model, the resource in a server is accessed by a client. On the other hand, web applications introduce a third party

to access the resource in a server. A resource owner allows a client to access the server. For example, you can allow Facebook to access your Gmail account to find your friends. In this case, Facebook is a client, Gmail is a server, and the contact lists in Gmail is your resource in the server. This environment is not uncommon in web applications and this kind of authentication is called delegated authentication. This delegated authentication is required for the client to access the resource in the server. The authentication protocol, OAuth 2.0 [52] has been proposed to support delegated authentication. Unfortunately, the security of OAuth 2.0 is based on the security of HTTPS [51], which is known to be vulnerable to Phishing and Pharming attacks. Therefore, a new authentication protocol, that is proven secure against Phishing and Pharming attacks, is needed to support delegated authentication.

Cloud Computing Environments: Cloud computing is an emerging new paradigm of computing [7]. Cloud computing has created new challenges for authentication because cloud computing involves both virtualization and delegation environments. In order to use the services in cloud computing, a new authentication protocol, that supports an integrated authentication for virtualization and delegated authentication, is needed.

Designing an authentication protocol is not a trivial task. As new technologies emerge and existing systems become more complicated, it becomes more challenging to design authentication protocols that support the new emerging environments.

Bibliography

[1] National strategy for trusted identities in cyberspace. Department of Homeland Security, June 2010.

[2] Ben Adida. SessionLock: Securing Web Sessions against Eavesdropping. In *WWW '08: Proceeding of the 17th international conference on World Wide Web*, pages 517–524, New York, NY, USA, 2008. ACM.

[3] Amitanand Aiyer, Lorenzo Alvisi, and Mohamed Gouda. Key grids: A protocol family for assigning symmetric keys. In *Proceedings of IEEE International Conference on Network Protocols (ICNP)*, pages 178–186, 2006.

[4] R. Arends, R. Austein, M. Larson, D. Massey, and S. Rose. DNS Security Introduction and Requirements. RFC 4033 (Proposed Standard), March 2005.

[5] R. Arends, R. Austein, M. Larson, D. Massey, and S. Rose. Protocol Modifications for the DNS Security Extensions. RFC 4035 (Proposed Standard), March 2005. Updated by RFC 4470.

[6] R. Arends, R. Austein, M. Larson, D. Massey, and S. Rose. Resource Records for the DNS Security Extensions. RFC 4034 (Proposed Standard), March 2005. Updated by RFC 4470.

[7] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica,

and Matei Zaharia. A view of cloud computing. *Commun. ACM*, 53(4):50–58, April 2010.

[8] D. Atkins and R. Austein. Threat Analysis of the Domain Name System (DNS). RFC 3833 (Informational), August 2004.

[9] Theodore Ts B. Clifford Neuman. Kerberos: An authentication service for computer networks. *IEEE Communications Magazine*, pages 33–44, September 1994.

[10] Bank of America. `http://www.bankofamerica.com/privacy/sitekey/`.

[11] Adam Barth, Collin Jackson, and John C. Mitchell. Robust defenses for cross-site request forgery. In *CCS '08:Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS 2008)*, 2008.

[12] Steven M. Bellovin and Michael Merritt. Encrypted key exchange: Password-based protocols secure against dictionary attacks. In *IEEE SYMPOSIUM ON RESEARCH IN SECURITY AND PRIVACY*, pages 72–84, 1992.

[13] Steven M. Bellovin and Michael Merritt. Augmented encrypted key exchange: A password-based protocol secure against dictionary attacks and password file compromise. In *ACM Conference on Computer and Communications Security*, pages 244–250, 1993.

[14] S. Blake-Wilson, M. Nystrom, D. Hopwood, J. Mikkelsen, and T. Wright. Transport Layer Security (TLS) Extensions. RFC 4366 (Proposed Standard), April 2006. Obsoleted by RFC 5246.

[15] David Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Communications of the ACM*, 24(2), February 1981.

[16] David Chaum. The dining cryptographers problem: Unconditional sender and recipient untraceability. *Journal of Cryptology*, 1:65–75, 1988.

[17] Sonia Chiasson, P. C. van Oorschot, and Robert Biddle. A usability study and critique of two password managers. In *USENIX-SS'06: Proceedings of the 15th conference on USENIX Security Symposium*, pages 1–1, Berkeley, CA, USA, 2006. USENIX Association.

[18] Taehwan Choi, Hrishikesh B. Acharya, and Mohamed G. Gouda. The best keying protocol for sensor networks. In *2nd IEEE International Workshop on Data Security and PrivAcy in wireless Networks*, pages 1–6, 2011.

[19] Taehwan Choi, Hrishikesh B. Acharya, and Mohamed G. Gouda. Is That You? Authentication in a Network without Identities. In *GLOBECOM*, pages 1–5. IEEE, 2011.

[20] Taehwan Choi, Hrishikesh B. Acharya, and Mohamed G. Gouda. TPP: The Two-Way Password Protocol. In Haohong Wang, Jin Li, George N. Rouskas, and Xiaobo Zhou, editors, *ICCCN*, pages 1–6. IEEE, 2011.

[21] Taehwan Choi and Mohamed G. Gouda. HTTPI: An HTTP with Integrity. In *The First International Workshop on Privacy, Security and Trust in Mobile and Wireless Systems (MobiPST 2011)*, 2011b.

[22] Taehwan Choi, Sooel Son, Mohamed G. Gouda, and Jorge Arturo Cobb. Pharewell to phishing. In Sandeep S. Kulkarni and André Schiper, editors, *SSS*, volume 5340 of *Lecture Notes in Computer Science*, pages 233–245. Springer, 2008.

[23] Neil Chou, Robert Ledesma, Yuka Teraguchi, Dan Boneh, and John C. Mitchell. Client-side defense against web-based identity theft. In *11th Annual Network and Distributed System Security Symposium (NDSS '04)*, 2004.

[24] Rachna Dhamija. Hash visualization in user authentication. In *Proceedings of the Computer Human Interaction 2000 Conference.*, April 2000.

[25] Rachna Dhamija and J. D. Tygar. The battle against phishing: Dynamic security skins. In *SOUPS '05: Proceedings of the 2005 symposium on Usable privacy and security*, pages 77–88, New York, NY, USA, 2005. ACM Press.

[26] Rachna Dhamija, J.D. Tygar, and Marti Hearst. Why phishing works. In *the Proceedings of the Conference on Human Factors in Computing Systems (CHI2006)*, 2006.

[27] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246 (Proposed Standard), August 2008.

[28] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: The second-generation onion router. In *Proceedings of the 13th USENIX Security Symposium*, August 2004.

[29] eBay Toolbar and Account Guard. `http://pages.ebay.com/help/confidence/account-guard.html`.

[30] Carl Ellison and Bruce Schneier. Ten Risks of PKI: What You're Not Being Told About Public Key Infrastructure. *Computer Security Journal*, 16(1):1–7, 2000.

[31] Ehab S. Elmallah, Mohamed G. Gouda, and Sandeep S. Kulkarni. Logarithmic keying. *ACM Transactions on Autonomic Systems*, 3:18:1–18:18, December 2008.

[32] P. Eronen and H. Tschofenig. Pre-Shared Key Ciphersuites for Transport Layer Security (TLS). RFC 4279 (Proposed Standard), December 2005.

[33] Laurent Eschenauer and Virgil D. Gligor. A key-management scheme for distributed sensor networks. In *Proceedings of the 9th ACM conference on Computer and communications security (CCS)*, pages 41–47, 2002.

[34] Edward W. Felten, Dirk Balfanz, Drew Dean, and Dan S. Wallach. Web spoofing: An internet con game. In *20th National Information Systems Security Conference (Baltimore, Maryland)*, October 1997.

[35] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard), June 1999. Updated by RFC 2817.

[36] Rob Franco. Website identification and extended validation certificates in IE7 and other browsers. IEBlog, Nov 2005.

[37] J. Franks, P. Hallam-Baker, J. Hostetler, S. Lawrence, P. Leach, A. Luotonen, and L. Stewart. HTTP Authentication: Basic and Digest Access Authentication. RFC 2617 (Draft Standard), June 1999.

[38] Michael J. Freedman and Robert Morris. Tarzan: A peer-to-peer anonymizing network layer. In *Proceedings of the 9th ACM Conference on Computer and Communications Security (CCS 2002)*, Washington, DC, November 2002.

[39] T. Freeman, R. Housley, A. Malpani, D. Cooper, and W. Polk. Server-Based Certificate Validation Protocol (SCVP). RFC 5055 (Proposed Standard), December 2007.

[40] Kevin Fu, Emil Sit, Kendra Smith, and Nick Feamster. Dos and don'ts of client authentication on the web. In *Proceedings of the 10th USENIX Security Symposium*, Washington, D.C., August 2001. An extended version is available as MIT-LCS-TR-818.

[41] Eran Gabber, Phillip B. Gibbons, Yossi Matias, and Alain Mayer. A convenient method for securely managing passwords. In *Financial Cryptography*, Feburuary 1997.

[42] Evgeniy Gabrilovich and Alex Gontmakher. The homograph attack. *Commun. ACM*, 45(2):128, 2002.

[43] Sandro Gauci. Surf Jacking - "HTTPS will not save you". `http://enablesecurity.com/2008/08/11/surf-jack-https-will-not-save-you`, August 2008.

[44] L. Gong and D. J. Wheeler. A matrix key-distribution scheme. *Journal of Cryptology*, 2:51–59, January 1990.

[45] Mohamed G. Gouda, Alex X. Liu, Lok M. Leung, and Mohamed A. Alam. Spp: An anti-phishing single password protocol. *Computer Networks*, 51(13):3715–3726, 2007.

[46] Robert Graham. Sidejacking with hamster. `http://erratasec.blogspot.com/2007/08/sidejacking-with-hamster_05.html`, August 2007.

[47] V. Griffith and M. Jakobsson. Messin' with Texas, Deriving Mother's Maiden Names using Public Records. In M. Yung J. Ioannidis, A. Keromytis, editor, *Applied Cryptography and Network Security: Third International Conference, Lecture Notes in Computer Science*, volume 3531. Springer Berlin / Heidelberg, June 2005.

[48] Anti-Phishing Working Group. Phising activity trends report. `http://www.antiphishing.org/reports/apwg_report_sept_2007.pdf`, Sept 2007.

[49] J. Alex Halderman, Brent Waters, and Edward W. Felten. A Convenient Method for Securely Managing Passwords. In *14th International World Wide Web Conference*, May 2005.

[50] E. Hammer-Lahav. The OAuth 1.0 Protocol. RFC 5849 (Informational), April 2010.

[51] Eran Hammer-Lahav. OAuth 2.0 and the Road to Hell.

[52] D. Hardt. The OAuth 2.0 Authorization Framework. Internet-Draft, February 2013.

[53] Amir Herzberg. Trustbar: Re-establishing trust in the web. `http://www.cs.biu.ac.il/~herzbea/TrustBar/`, 2006.

[54] Andrew Hintz. Fingerprinting websites using traffic analysis. In *Proceedings of the 2nd international conference on Privacy enhancing technologies*, PET'02, pages 171–178, Berlin, Heidelberg, 2003. Springer-Verlag.

[55] R. Housley, W. Polk, W. Ford, and D. Solo. Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. RFC 3280 (Proposed Standard), April 2002. Obsoleted by RFC 5280, updated by RFCs 4325, 4630.

[56] Andrew Howard, Maja J Mataric, and Gaurav S Sukhatme. Mobile sensor network deployment using potential fields: A distributed, scalable solution to the area coverage problem. In *Proceedings of the International Symposium on Distributed Autonomous Robotic Systems (DARS)*, pages 299–308, 2002.

[57] Sean Hynes and Neil C. Rowe. A multi-agent simulation for assessing massive sensor deployment. *Journal of Battlefield Technology*, 7:23–36, 2004.

[58] David P. Jablon. Strong password-only authenticated key exchange. *Computer Communication Review*, 26(5):5–26, 1996.

[59] Collin Jackson and Adam Barth. ForceHTTPS: protecting high-security web sites from network attacks. In *WWW '08: Proceeding of the 17th international*

conference on World Wide Web, pages 525–534, New York, NY, USA, 2008. ACM.

[60] Collin Jackson, Adam Barth, Andrew Bortz, Weidong Shao, and Dan Boneh. Protecting Browsers from DNS Rebinding Attacks. In *In Proceedings of ACM CCS 07*, 2007.

[61] Collin Jackson, Daniel R. Simon, Desney S. Tan, and Adam Barth. An evaluation of extended validation and picture-in-picture phishing attacks. In *Usable Security*, 2007.

[62] Paul Johnston and Richard Moore. Multiple browser cookie injection vulnerabilities. `http://www.westpoint.ltd.uk/advisories/wp-04-0001.txt`, September 2004.

[63] Ari Juels, Markus Jakobsson, and Tom N. Jagatic. Cache cookies for browser authentication (extended abstract). In *SP '06: Proceedings of the 2006 IEEE Symposium on Security and Privacy (S&P'06)*, pages 301–305, Washington, DC, USA, 2006. IEEE Computer Society.

[64] Ari Juels, Markus Jakobsson, and Sid Stamm. Active cookies for browser authentication. In *14th Annual Network and Distributed System Security Symposium (NDSS '07)*, 2007.

[65] Chris Karlof, Umesh Shankar, J. Doug Tygar, and David Wagner. Dynamic pharming attacks and locked same-origin policies for web browsers. In *ACM Conference on Computer and Communications Security*, pages 58–71, 2007.

[66] Charlie Kaufman and Radia Perlman. PDM: a new strong password-based protocol. In *SSYM'01: Proceedings of the 10th conference on USENIX Security Symposium*, pages 23–23, Berkeley, CA, USA, 2001. USENIX Association.

[67] John Kelsey, Bruce Schneier, Chris Hall, and David Wagner. Secure applications of low-entropy keys. In *ISW '97: Proceedings of the First International Workshop on Informat ion Security*, pages 121–134, London, UK, 1998. Springer-Verlag.

[68] S. Kent. IP Authentication Header. RFC 4302 (Proposed Standard), December 2005.

[69] S. Kent. IP Encapsulating Security Payload (ESP). RFC 4303 (Proposed Standard), December 2005.

[70] S. Kent and K. Seo. Security Architecture for the Internet Protocol. RFC 4301 (Proposed Standard), December 2005.

[71] Amit Klein. Cross site scripting explained. Sanctum Security Group, June 2002.

[72] Mitja Kolšek. Session fixation vulnerability in web-based applications. `www.acros.si/papers/session_fixation.pdf`, December 2002.

[73] H. Krawczyk, M. Bellare, and R. Canetti. HMAC: Keyed-Hashing for Message Authentication. RFC 2104 (Informational), February 1997.

[74] Sandeep S. Kulkarni, Mohamed G. Gouda, and Anish Arora. Secret instantiation in ad-hoc networks. *Special Issue of Elsevier Journal of Computer Communications on Dependable Wireless Sensor Networks*, 29:200–215, 2005.

[75] Leslie Lamport. Password authentication with insecure communication. *Commun. ACM*, 24:770–772, November 1981.

[76] Moxie Marlinspike. New Techniques for Defeating SSL/TLS. `http://www.blackhat.com/presentations/bh-dc-09/Marlinspike/BlackHat-DC-09-Marlinspike-Defeating-SSL.pdf`, February 2009.

[77] Chris Masone, Kwang-Hyun Baek, and Sean Smith. WSKE: Web server key enabled cookies. In *Usable Security (USEC)*, 2007.

[78] Rober McMillan. Gartner: Consumers to lose $2.8 billion to phishers in 2006. `http://www.networkworld.com/news/2006/110906-gartner-consumers-to-lose-28b.html`, 2006.

[79] Michael Atighetchi and Partha Pal. PhishBouncer: An HTTPS proxy for attribute-based prevention of Phishing Attacks. In *submitted to The second APWG eCrime Researchers Summit*, 2007.

[80] Microsoft. Erroneous verisign issued digital certificates pose spoofing hazard. Technical Report Microsoft Security Bulletin MS01-017, 2001.

[81] Microsoft. .net passport, 2010.

[82] Alan Mislove, Gaurav Oberoi, Ansley Post, Charles Reis, Peter Druschel, and Dan S. Wallach. AP3: cooperative, decentralized anonymous communication. In *EW 11: Proceedings of the 11th workshop on ACM SIGOPS European workshop*, page 30, New York, NY, USA, 2004. ACM.

[83] Prateek Mittal and Nikita Borisov. Information leaks in structured peer-to-peer anonymous communication systems. In *CCS '08: Proceedings of the 15th ACM conference on Computer and communications security*, pages 267–278, New York, NY, USA, 2008. ACM.

[84] J. Mogul and A. Van Hoff. Instance Digests in HTTP. RFC 3230 (Proposed Standard), January 2002.

[85] Jeffery C. Mogul, Yee Man Chan, and Terence Kelly. Design, implementation, and evaluation of duplicate transfer detection in http. In *NSDI'04: Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation*, pages 4–4, Berkeley, CA, USA, 2004. USENIX Association.

[86] J. Myers and M. Rose. The Content-MD5 Header Field. RFC 1864 (Draft Standard), October 1995.

[87] M. Myers, R. Ankney, A. Malpani, S. Galperin, and C. Adams. X.509 Internet Public Key Infrastructure Online Certificate Status Protocol - OCSP. RFC 2560 (Proposed Standard), June 1999.

[88] Arjun Nambiar and Matthew Wright. Salsa: A structured approach to large-scale anonymity. In *Proceedings of CCS 2006*, October 2006.

[89] Netcraft. `http://www.netcraft.com`.

[90] Ola Nordström and Constantinos Dovrolis. Beware of BGP attacks. *SIGCOMM Comput. Commun. Rev.*, 34(2):1–8, 2004.

[91] Mark Nottingham. Inherent HTTP Coherence. `http://www.mnot.net/papers/coherence.html`.

[92] Gunter Ollmann. The pharming guide. `http://www.ngssoftware.com/papers/ThePharmingGuide.pdf`.

[93] OpenDNS. `http://www.opendns.com`.

[94] PhishTank. `http://www.phishtank.com`, Nov 2006.

[95] D. Pinkas and R. Housley. Delegated Path Validation and Delegated Path Discovery Protocol Requirements. RFC 3379 (Informational), September 2002.

[96] Zulfikar Ramzan. DNS Pharming Attacks Using Rogue DHCP. `http://www.symantec.com/connect/blogs/dns-pharming-attacks-using-rogue-dhcp`, December 2008.

[97] Zulfikar Ramzan. Drive-by pharming in the wild. `http://www.symantec.com/connect/blogs/drive-pharming-wild`, January 2008.

[98] Charles Reis, Steven D. Gribble, Tadayoshi Kohno, and Nicholas C. Weaver. Detecting in-flight page changes with web tripwires. In *NSDI'08: Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, pages 31–44, Berkeley, CA, USA, 2008. USENIX Association.

[99] Michael Reiter and Aviel Rubin. Crowds: Anonymity for web transactions. *ACM Transactions on Information and System Security*, 1(1), June 1998.

[100] E. Rescorla. HTTP Over TLS. RFC 2818 (Informational), May 2000.

[101] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM conference on Computer and communications security*, CCS '09, pages 199–212, New York, NY, USA, 2009. ACM.

[102] Blake Ross, Collin Jackson, Nick Miyake, Dan Boneh, and John C. Mitchell. Stronger password authentication using browser extensions. In *SSYM'05: Proceedings of the 14th conference on USENIX Security Symposium*, pages 2–2, Berkeley, CA, USA, 2005. USENIX Association.

[103] Aviel D. Rubin. Independent one-time passwords. In *Proceedings of the 5th conference on USENIX UNIX Security Symposium - Volume 5*, pages 15–15, 1995.

[104] Roi Saltzman and Adi Sharabani. Active man in the middle attacks. OWASP AU 2009, Feburuary 2009.

[105] S. Sana and M. Matsumoto. A wireless sensor network protocol for disaster management. In *Proceedings of Information, Decision and Control (IDC)*, pages 209 –213, 2007.

[106] Stuart E. Schechter, Rachna Dhamija, Andy Ozment, and Ian Fischer. The emperor's new security indicators. In *SP '07: Proceedings of the 2007 IEEE Symposium on Security and Privacy*, pages 51–65, Washington, DC, USA, 2007. IEEE Computer Society.

[107] Bruce Schneier. Semantic attacks: The third wave of network attacks. Crypto-Gram Newsletter, Sept 2006.

[108] PassMark Security. `http://www.passmarksecurity.com`.

[109] C. Shields. Responder anonymity and anonymous peer-to-peer file sharing. In *Proceedings of the Ninth International Conference on Network Protocols*, pages 272–, Washington, DC, USA, 2001. IEEE Computer Society.

[110] SpoofStick. `http://www.spoofstick.com`.

[111] Sid Stamm, Zulfikar Ramzan, and Markus Jakobsson. Drive-by pharming. In *ICICS*, pages 495–506, 2007.

[112] Paul F. Syverson, David M. Goldschlag, and Michael G. Reed. Anonymous connections and onion routing. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, SP '97, pages 44–, Washington, DC, USA, 1997. IEEE Computer Society.

[113] David Taylor, Thomas Wu, Nikos Mavroyanopoulos, and Trevor Perrin. Using SRP for TLS Authentication. IETF TLS Working Group: http://www.ietf.org/internet-draft/draft-ietf-tls-srp-14.txt, December 2007.

[114] Trusteer. Measuring the effectiveness of in-the-wild phishing attacks, 2009.

[115] US-CERT. Multiple DNS implementations vulnerable to cache poisoning. `http://www.kb.cert.org/vuls/id/800113`.

[116] Alma Whitten and J. D. Tygar. Why Johnny can't encrypt: A usability evaluation of PGP 5.0. In *SSYM'99: Proceedings of the 8th conference on USENIX Security Symposium*, pages 14–14, Berkeley, CA, USA, 1999. USENIX Association.

[117] Min Wu, Robert C. Miller, and Simson L. Garfinkel. Do security toolbars actually prevent phishing attacks? In *CHI '06: Proceedings of the SIGCHI conference on Human Factors in computing systems*, pages 601–610, New York, NY, USA, 2006. ACM Press.

[118] Min Wu, Robert C. Miller, and Greg Little. Web wallet: Preventing phishing attacks by revealing user intentions. In *SOUPS '06: Proceedings of the second symposium on Usable privacy and security*, pages 102–113, New York, NY, USA, 2006. ACM Press.

[119] Thomas Wu. Srp-6: Improvements and refinements to the secure remote password protocol. Submission to the IEEE P1363 Working Group.

[120] Thomas Wu. The Secure Remote Password Protocol. In *Proceedings of the 1998 Internet Society Network and Distributed System Security Symposium*, pages 97–111, 1998.

[121] Yahoo. `http://help.yahoo.com/l/us/yahoo/edit/privacy/edit-39.html`.

[122] Ka-Ping Yee and Kragen Sitaker. Passpet: convenient password management and phishing protection. In *SOUPS '06: Proceedings of the second symposium on Usable privacy and security*, pages 32–43, New York, NY, USA, 2006. ACM.

[123] Jim Youll. Fraud Vulnerabilities in SiteKey Security at Bank of America. Technical report, Challenge/Response,LLC, July 2006.

[124] Bojan Zdrnja. Massive ARP spoofing attacks on web sites. `http://isc.sans.org/diary.html?storyid=6001`, March 2009.

[125] William Zeller and Edward W. Felten. Cross-site request forgeries: Exploitation and prevention. Technical report, Department of Computer Science, Center for Information Technology Policy, Princeton University, October 2008.

[126] Kai Zhang. ARP spoofing HTTP infection malware. `http://securitylabs.websense.com/content/Blogs/2885.aspx`, December 2007.

[127] Yue Zhang, Serge Egelman, Lorrie Faith Cranor, and Jason Hong. Phinding phish: Evaluating anti-phishing tools. In *Proceedings of the 14th Annual Network & Distributed System Security Symposium (NDSS 2007)*, 2007.

[128] Yue Zhang, Jason Hong, and Lorrie Faith Cranor. CANTINA: A content-based approach to detecting phishing web sites. In *Proceedings of the 16th International conference on World Wide Web*, 2007.

[129] Li Zhuang, Feng Zhou, Ben Y. Zhao, and Antony I. T. Rowstron. Cashmere: Resilient anonymous routing. In *NSDI*, 2005.

Vita

Taehwan Choi received a BS in Math from Sogang University in Korea in 1998 and received a MS in Computer Science from Yonsei University in Korea in 2002. His former research interests were networking and focused on Mobile Networking. His current research interest is security in general and he is interested in web security, network security, platform security, and software security. He is interested in designing a secure system architecture, with which he can solve challenging security problems of the Internet. He worked as a software engineer for POSCO ICT (formerly POSDATA) from 1998 to 2000 and as a research engineer for LG Electronics from 2002 to 2005.

Email Address: ctlight@gmail.com

This dissertation was typeset with LaTeX 2_ε[1] by the author.

[1] LaTeX 2_ε is an extension of LaTeX. LaTeX is a collection of macros for TeX. TeX is a trademark of the American Mathematical Society. The macros used in formatting this dissertation were written by Dinesh Das, Department of Computer Sciences, The University of Texas at Austin, and extended by Bert Kay, James A. Bednar, and Ayman El-Khashab.